Aprendendo a
PROGRAMAR
LINGUAGEM SWIFT
Tito Petri

Primeira edição, 2022
ISBN 0-5487263-1-5
www.titopetri.com.br

ÍNDICE

1. O Que Aprenderemos neste Livro?

Neste material você vai aprender sobre **Programação de *Softwares***, começando do zero, até criar suas **Sete Primeiras Aplicações**, com a **Linguagem Swift**.

O objetivo deste Livro é fazer com que o aluno aprenda a parte teórica da **Lógica de Programação**, conhecendo os principais termos, conceitos e a técnica de se criar algoritmos e programas de computador.

Quando falamos sobre **Códigos de Programação**, muitos pensam que a parte teórica não é muito importante e acabam apenas decorando ou copiando códigos.

Através deste Livro, vou te ensinar que programar é o mesmo que criar um método para **solucionar algum problema**. Esta é uma grande habilidade de raciocínio que você levará para sua vida.

Ao final deste material, você terá adquirido novos conhecimentos em:

- Lógica de Programação
- Criação de Algoritmos
- Linguagem de Programação Swift
- Criação do seu Primeiro Aplicativo iOS

Diga "Olá!" para o **Felpudo**!

O Passarinho Desenvolvedor mais *Geek*, *Nerd e* Intelectual que você jamais conheceu!

Felpudo

O Felpudo vai nos ensinar tudo sobre programação, passando as idéias e conceitos de forma ilustrada, lúdica e divertida.

Este Livro busca trazer uma experiência nova e mais divertida para o aprendizado, para que um jovem ou até uma criança aprenda a programar e adquira um maior gosto pela prática.

Criaremos **sete Projetos** desafiadores, para treinar a técnica de desenvolver programas com a Linguagem Swift, desde o conceito inicial, até a aplicação funcionando.

Para isso, vamos utilizar o Xcode, o programa desenvolvido pela Apple para criar aplicações para seus dispositivos Mac OS (computadores e notebooks), iOS (iPhones e iPads), tvOS (Apple TV) e até watchOS (Apple Watch).

Ao final do material, também vou te ensinar a **Instalar** e dar os **Primeiros Passos** no **Xcode IDE**, para que você crie o seu **Primeiro Aplicativo iOS** utilizando o Swift.

O conhecimento adquirido com este material, vai te dar um ótimo embasamento para aprender qualquer outra linguagem de programação como **Java, C++, Swift**, **C# (C-Sharp)**, **Python** ou **JavaScript**.

Programar é uma valiosa habilidade que nos próximos anos se aplicará em praticamente todos os campos e áreas de atuação.

Lembre-se de que os conhecimentos em Lógica e Algoritmo, são aplicados ao desenvolvimento de **todo tipo** de **Aplicação Digital**, como em ***Videogames***, **Criação de *Sites***, **Sistemas**, **Produtos Eletrônicos**, **Robótica**, e outros!

2. Linguagem de Programação Swift

Swift é a Linguagem criada pela Apple Computers, para que os desenvolvedores possam criar aplicações para seus dispositivos. É uma linguagem poderosa e muito fácil de entender, pois o código é bem simplificado, além do Xcode nos dar boas dicas e pistas enquanto estamos programando.

Acompanhando este material, você vai aprender rápida e metodicamente tudo sobre o fundamento da **Lógica de Programação**, já aplicada à linguagem **Swift**.

Criaremos juntos **Sete Programas Completos em Swift** enquanto aprendemos Swift e a Lógica de Programação. Veja abaixo os algoritmos que iremos desenvolver:

1. Par ou Ímpar
2. Jokempô
3. Sorteio de Nomes
4. Buscar Nome
5. Validação do CPF
6. Números Primos
7. Criar e Consultar Cadastro

3. Entenda bem este Material

Antes de sair escrevendo as linhas de código, é muito importante entender alguns conceitos fundamentais da programação em geral.

Preste atenção nestes pontos para conseguir entender e absorver bem este material.

a) Quadro de Exemplos de Código

Todos os códigos que aparecem pela primeira vez, estarão destacados em **negrito** dentro do quadro de exemplo.

Exemplo:

```
print("Olá! Vamos Programar em Swift!")

let pais = "Brasil"
```

Os códigos que aparecerão no quadro de exemplo procuram mostrar apenas os novos comandos, que dificilmente vão se repetir.

Comandos referentes ao assunto que estamos abordando, aparecerão sempre em **negrito**.

b) Códigos e Projetos para Baixar

Todos os códigos e projetos desenvolvidos durante este Livro estarão disponíveis através do *link* abaixo:

http://www.titopetri.com.br/recursos/AnexosLivroSwift.zip

c) Curso Online de iOS com Swift

Este Livro também foi desenvolvido em formato de videoaulas *online*. Caso você queira, poderá assistir às videoaulas, enquanto acompanha por este Livro.

Acesse a página do curso online de programação em Swift e aprenda junto com o Livro a programar. Use este como seu material de apoio.

4. Compilador Online de Swift

Existem duas maneiras diferentes de programar em Swift.

A Primeira é utilizando o Xcode IDE, a ferramenta oficial da Apple para criar aplicativos. Só funciona no MacOS.

Lembre-se que se você quiser se tornar um Desenvolvedor iOS, inevitavelmente terá um dia que adquirir um computador Mac e um iPhone.

Porém, por enquanto, se você ainda não possui um computador Mac ou Sistema MacOS, pode utilizar um compilador online de Swift. Segue uma sugestão:

https://www.tutorialspoint.com/compile_swift_online.php

Lembrando que este compilador utiliza uma versão anterior do Swift (4) e possui várias limitações.

Não é possível gerar uma aplicação para iOS ou MacOS através do compilador online. Somente utilizando o Xcode IDE, que vamos conhecer adiante.

5. Xcode IDE

O Xcode IDE (*Integrated Development Environment* ou *Ambiente de Desenvolvimento Integrado*) é o programa capaz de juntar a Linguagem Swift com qualquer outro tipo de recurso gráfico (Imagens, Animações, Vídeos) e de funcionalidades (*Internet*, Banco de Dados e todo tipo de recurso dos iPhones e iPads, como *Internet*, Câmera, Acelerômetro ou Geolocalização).

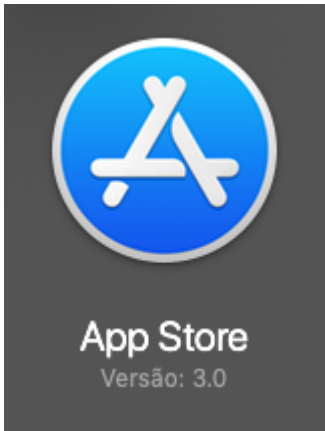
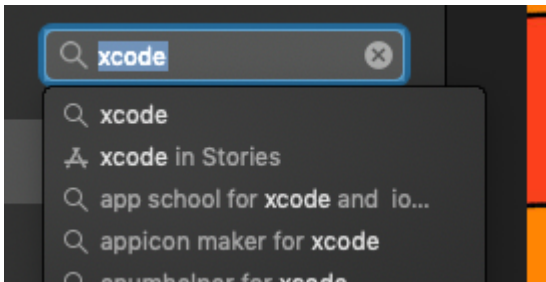
Porém, o IDE requer muito mais conhecimento para ser utilizado. Recomendo sempre aos alunos tentarem entender antes a Linguagem Swift (princípios, regras e fundamentos) e depois partirem para o Xcode (para construir apps e trabalhar com toda a parte visual).

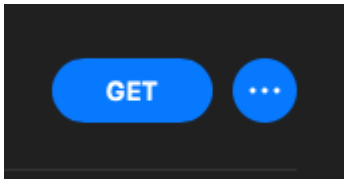
Atualmente o Xcode está na versão 10.2.1 (de 17 de abril de 2019) e você pode baixá-lo gratuitamente através da appStore.

Basta ter uma conta de usuário da Apple (a mesma que utilizamos para baixar aplicativos iOS da appStore).

6. Download e Instalação do Xcode

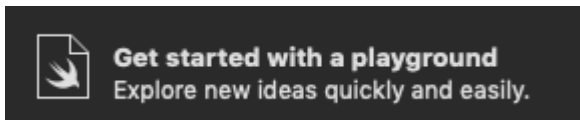
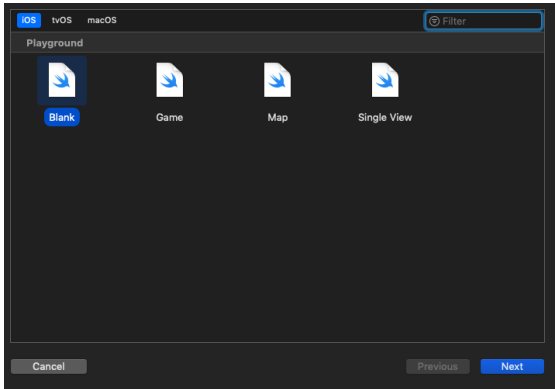
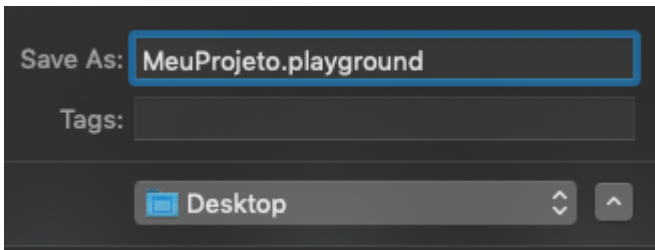
Lembre-se então que o requisito mínimo para instalar o Xcode, é o Sistema Operacional Mac OS.

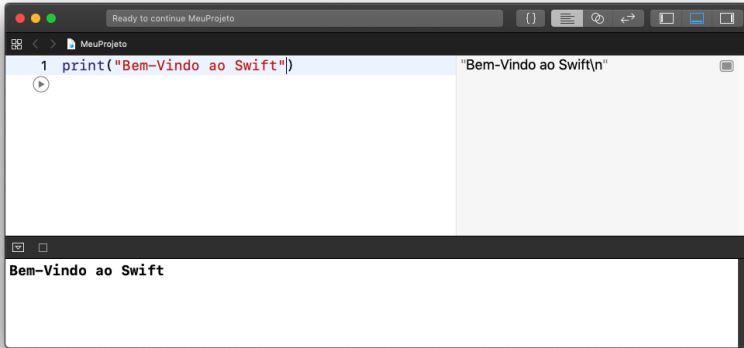
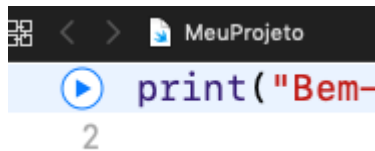
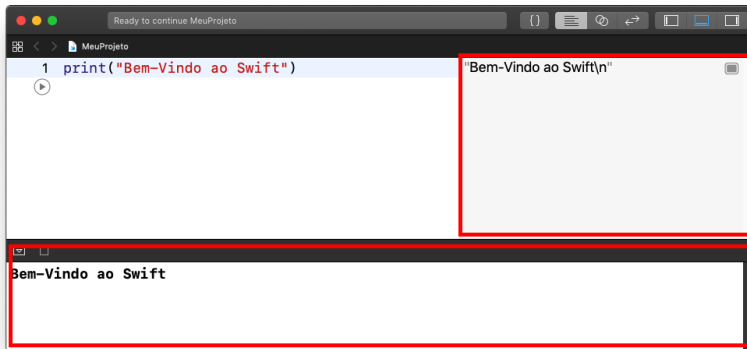
Abra o aplicativo App Store

No campo de busca por aplicativos, digite ***xcode***

Caso seja necessário, faça o *login* com a sua conta da Apple.

Clique em **Obter / Get** para fazer o download e a instalação do Xcode.

7. Escrevendo o Primeiro Código

Agora que já temos todos os conhecimentos preliminares sobre o Swift, e já fizemos o Download e Instalação do Xcode, vamos escrever nosso primeiro programa.

Abra o Xcode. Aparecerão algumas opções para iniciar novo projeto ou abrir projetos recentes.

Vamos começar brincando de programação com o Playground. Clique na opção *Get Started with Playground*.

Selecione a opção *Blank* e clique em *Next*.
Depois dê um nome qualquer para seu projeto e salve na sua pasta.

O Playground é um ambiente bem simplificado que iremos utilizar para escrever nossos primeiros programas em Swift. É a opção ideal para quem ainda não conhece a linguagem e está tentando entender os primeiros códigos e comandos ou até para prototipar e testar algum código ou algoritmo.

Abra o Playground e espere até que tudo seja carregado, isso pode demorar até alguns minutos.

Escreva o mesmo exemplo que vimos anteriormente, e clique no botão Play que vai aparecer ao lado da linha de comando.

Perceba que ao executar o código, o Playground vai te mostrar dois resultados, um no menu lateral direito, onde é exibido o resultado de cada linha de comando.

O outro na barra inferior, onde é exibido o resultado final do código executado. Mais adiante entenderemos isso melhor.

Por enquanto tudo que precisamos entender é que para programar em swift, basta escrever os códigos e compila-los no Playground para ver o resultado do nosso algoritmo.

8. Regras Básicas para Programar

Primeiro, precisamos entender algumas regras básicas do Swift e de programação em geral para entender 100% do que está escrito neste código. Comece memorizando estas informações:

a) Case Sensitive

O Swift é ***Case Sensitive***, ou seja, ele leva em consideração a diferença entre a letra maiúscula e a letra minúscula. Veja o seguinte comando para exibir um valor no console.

Exemplo:

```
print("Bem-Vindo ao Xcode e Swift")
```

Nunca poderá ser escrito como **PRINT** ou **Print.** Isto ocasionará erros na sua aplicação.

b) Aspas Duplas

Para escrever uma **palavra**, **frase** ou **cadeia de caracteres** (letras) devemos iniciar e terminar sempre com **aspas duplas** (**"**).

Agora, baseando-se nas idéias obtidas, leia atenciosamente o comando a seguir e perceba os conceitos envolvidos em uma única linha de código.

Exemplo:

```
print("Bem-Vindo ao Xcode e Swift")
```

Vamos entender palavra por palavra o que diz nosso código.

O comando **print()** serve para exibir uma mensagem no console.

Exemplo:

```
print("Oi! Eu sou o Felpudo!")
print(12345)
```

Entre os parênteses passamos o argumento para este comando. Ou seja, o objeto ou valor que deve ser exibido.

Exemplo:

```
print( "Bem-Vindo ao Swift!" )
```

Ao passar por argumento uma frase, ou cadeia de caracteres (que chamamos de String) então encontra-se entre aspas duplas.

Para indicar o fim de uma linha de comando, utilizamos o **ponto e vírgula** (**;**) após o comando.

No Swift, este comando é opcional, ao encontrar um **parágrafo**, o compilador já sabe que chegou ao fim da instrução.

São algumas idéias como essa que fazem o Swift ficar mais fácil de escrever do que as outras linguagens.

Exemplo:

```
print("Bem-Vindo ao Xcode e Swift")
```

É o mesmo que:

Exemplo:

```
print("Bem-Vindo ao Xcode e Swift")
```

Viu? Cada caractere, letra, número, vírgula ou ponto tem um porquê na programação.

Ao criar um código, você deve saber exatamente o que cada letrinha está querendo dizer.

Agora solucione estes **Desafios** para provar o que você aprendeu:

DESAFIO #1

Eu te desafio a fazer um Programa em Swift que imprima o seu nome completo no Console de Mensagens.

Faça-o da maneira mais simples, utilizando o mínimo possível de linhas de código.

Lesmo

DESAFIO #2

Bugado

Pesquise sobre as Origens da Linguagem Swift.

Quando e Onde ela foi criada?

Qual foi a empresa ou programador(es) que a criaram?

Antes de passar para a próxima fase, certifique-se de que aprendeu e experimentou de verdade conhecimentos mencionados nos desafios acima.

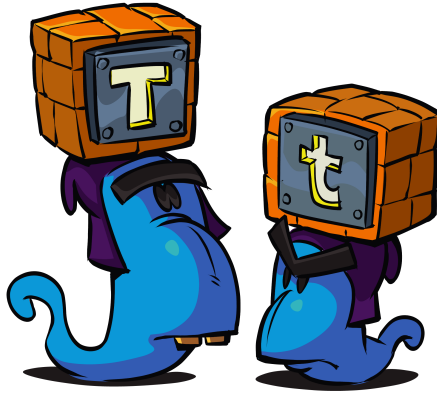
Você terá que utilizá-los o tempo todo!

Compartilhe este conhecimento nas suas redes sociais e mostre aos seus amigos o que você aprendeu de novo!

c) Caracteres com Acentuação

Alguns compiladores assim como o Xcode, não funcionam com determinados **Caracteres ASCII da Língua Portuguesa** (caracteres com acento principalmente).

d) Camel Case

É a prática de escrever palavras ou frase compostas de modo que cada palavra comece com uma letra maiúscula, sem espaços ou pontuação intermediários.

Isto facilita a leitura do código, e tornou-se um padrão entre os bons programadores.

Exemplo:

eusouumavariavel ou **EuSouUmaVariavel**

Perceba que no "Caso do Camelo" a frase fica muito mais legível.

Então lembre-se de usar esta boa prática na hora de criar os nomes para suas **classes**, **funções** e **variáveis** na programação.

Outra regra que vale lembrar, é que existe o **lowerCamelCase**, onde a primeira letra inicia em **minúsculo** (geralmente usamos para declarar **variáveis** ou **métodos**).

Exemplos comuns incluem "iPhone" e "eBay".

E o **UpperCamelCase** com apenas a primeira letra iniciada em **maiúsculo** (usada geralmente para declarar **classes**).

Não se preocupe pois entenderemos adiante o que são estas **variáveis**, **métodos** e **classes**.

e) Limites ou Escopos

Escopo significa um local bem determinado com o intento de se atingir. Significa o limite ou abrangência de uma operação.

Na programação, definimos os **escopos** ou **limites** de um **método** ou **classe** utilizando o caractere das **chaves {...}**.

Por enquanto, apenas perceba que no trecho de código, existem três objetos um contido no outro, estes três objetos são definidos ao **abrir e fechar as chaves {...}.**

```
78
79 func buscaContato(contatos:[Contato], nomeBuscado:String){
80     var encontrado:Bool = false
81     for item in contatos {
82         let meuContato:Contato = item as Contato
83
84         if meuContato.nome! == nomeBuscado {
85             print("Nome Encontrado!!!\n...")
86             print(meuContato.nome!)
87             print(meuContato.email!)
88             print(meuContato.telefone!)
89             encontrado = true
90         }
91     }
92
93     if !encontrado {
94         print("Nome Não Encontrado!")
95     }
96 }
97
```

Através deste conceito de escopos, estruturamos a nossa programação em um conceito de **hierarquia** ou **parentesco** que também define a **visibilidade** e **acesso** à estes objetos.

Por enquanto, entenda que basicamente quando você abre e fecha uma chave na programação, você está criando esses blocos de objetos.

E lembre-se de que sempre em que você abrir uma chave, deverá depois fechá-la.

6. Variáveis no Swift

Para programar, precisamos manipular valores ou dados que são armazenados em "recipientes" identificados.

Imagine uma caixa com algum objeto guardado nela. Esta caixa possui um nome para você se lembrar o que ela contém.

É claro que a caixa não existe, ela representa os **espaços na memória** do dispositivo.

Para o computador, esta variável tem um endereço numérico que indica a posição na memória.

Na nossa programação vamos dar nomes às variáveis, que representam estes endereços.

Pense que a Memória RAM do computador é um armário, separado em vários recipientes, cada um com um nome (Variável). E, em cada recipiente você pode guardar um determinado tipo de item (Dado ou Valor).

Veja abaixo alguns dos principais **Tipos de Dados** que podemos criar e utilizar na programação em geral.

TIPO	DESCRIÇÃO	EXEMPLO
Bool	Verdadeiro ou Falso	true, false
Int	Números Inteiros	7, -10, 500
Float	Números Decimais	1.5f, 0.75f, -5.5f
Double	Números Decimais	100.0, 5.7, 1.0, 3.2
String	Palavras e Frases	"Tito","Felpudo","Fofura"

Cada variável só pode ser de um único tipo, e só pode armazenar objetos daquele tipo.

Por exemplo, você nunca poderá guardar um nome em uma variável que já foi numérica.

Para criarmos uma variável na programação em Swift, precisamos seguir uma regra.

Veja o exemplo da criação de uma variável do tipo String, para armazenar um nome.

Exemplo:

```
var nome = "Tito Petri"
```

Perceba que precisamos seguir uma regra para dizer para o compilador que **tipo de dado** estamos criando, **qual o nome** desta variável, e a sua **inicialização**, que é o valor que será guardado na variável.

Para criar uma variável em Swift, você deve primeiro escrever a palavra var

O segundo argumento é o nome da variável que você vai criar

Seguindo com : você diz qual é o tipo desta variável

Exemplo:

```
var nome:String
var idade:Int
var peso:Float
```

No momento da declaração, podemos ou não inicializar a variável, ou guardar um valor dentro dela.

Exemplo:

```
var nome:String = "Tito"
var sobrenome = "Petri"
```

Perceba as duas declarações anteriores, podemos inicializar as variáveis com algum valor.

Repare que a declaração do tipo da variável também é opcional, pois ao guardar a palavra "Petri" na variável, o Swift já entendeu que o tipo é um String.

Neste momento da **Declaração das Variáveis,** precisamos entender mais algumas regras:

1) Podemos usar qualquer nome para as variáveis. Contando que o nome respeite o seguinte:

- Não seja uma **Palavra Reservadas,** comandos da linguagem.
(Exemplo: print, Float, String, Class, func, e muitas outras)

- Não pode conter em **Caractere Especial**.
(Exemplo: ! @ #$%ˆ&*)

- Não pode **Iniciar em Número**.
(Exemplo: 50nome, 123, 1idade)

2) É possível declarar a variável sem inicializá-la de imediato.

Escreva o seu **Nome** e o seu **Tipo** e, mais adiante, atribua à ela um valor no código.

Exemplo:

```
var nome:String

nome = "Tito Petri e Felpudo"
```

3) O símbolo de igual (**=**) neste momento não tem o sentido de **comparação** e sim de **atribuição**. Ou seja, estamos atribuindo, ou guardando um valor em uma determinada variável.

Exemplo:

```
var idade = 12
```

No exemplo acima, criamos uma variável do tipo **Int**, damos a ela o nome de **idade** e guardamos nela o valor **12**.

4) Os números decimais são representados por dois tipos diferentes (**Float** ou **Double**).

A diferença entre estes dois tipos, é que o **Double** armazena o dobro de casas decimais depois do ponto (o que garante uma maior precisão do número, ideal para aplicações gráficas)

Consequentemente, o **Float** é mais leve e ocupa menos espaço na memória.

Exemplo:

```
var idade:Int = 30
var altura:Float = 1.85
var peso:Double = 80.0
var ligado:Bool = true
```

7. Constantes no Swift

Existem também as constantes. Objetos declarados que possuem um valor fixo.

Após declarar uma constante, o seu valor não poderá mais ser modificado.

Para declarar uma constante em Swift,, utilizamos a palavra **let** ao invés da palavra **var**.

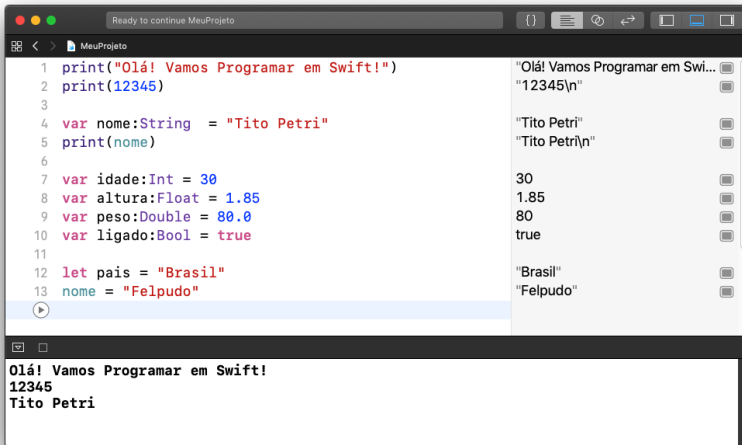
Declare algumas **variáveis** e **constantes** e execute o seu código.

Exemplo:

```
print("Olá! Vamos Programar em Swift!")
print(12345)

var nome:String  = "Tito Petri"
print(nome)

var idade:Int = 30
var altura:Float = 1.85
var peso:Double = 80.0
var ligado:Bool = true

let pais = "Brasil"
nome = "Felpudo"
```

Faça alguns testes, brinque com o código. Perceba que as constantes não podem ter o seu valor alterado, já as variáveis sim.

```
print("Olá! Vamos Programar em Swift!")
print(12345)

var nome:String  = "Tito Petri"
print(nome)

var idade:Int = 30
var altura:Float = 1.85
var peso:Double = 80.0
var ligado:Bool = true

let pais = "Brasil"
nome = "Felpudo"
```

```
Olá! Vamos Programar em Swift!
12345
Tito Petri
```

Execute novamente a aplicação, e se tudo estiver correto, você verá os valores declarados aparecerem na Barra do Console Lateral (onde os valores aparecem já ao serem declarados) e na barra Inferior (comandos **print**).

7. Comentando o Código

Existe a possibilidade de comentar linhas ou pedaços do código que está escrevendo.

O Compilador irá ignorar a parte comentada na compilação e execução do programa.

No caso do Swift, utilizamos os seguintes operadores:

//	Comenta a linha atual.
/*	Inicia um comentário em múltiplas linhas.
***/**	Encerra um comentário em múltiplas linhas.

Os comentários servem para você **descrever** e **documentar** um determinado trecho do código ou apenas **desativar algum comando**.

Esta documentação vale tanto para você mesmo entender o que está programando, quanto para outros programadores que eventualmente venham a usar o seu código ou projeto.

Exemplo:

```
// Utilizando o Playground

print("Olá! Vamos Programar em Swift!")
print(12345)

// Declaração de Variável

var nome:String  = "Tito Petri"
print(nome)

// Tipos de Variáveis
var idade:Int = 30
var altura:Float = 1.85
var peso:Double = 80.0
var ligado:Bool = true

/*
  Início do Bloco Comentado
.
.

.
  Fim do Bloco Comentado
*/

nome = "Felpudo"
```

8. Concatenação de Valores

Concatenar é o mesmo que **juntar** valores. Quando trabalhamos com frases ou Strings, podemos adicionar ao String valores do conteúdo de outras variáveis ou constantes. Observe o exemplo a seguir:

Exemplo:

```
let nome = "Felpudo Jr."
let idade = 12
let altura = 0.55

print("Nome: \(nome) - Idade: \(idade) - Altura: \(altura)")
```

Perceba que declaramos 3 variáveis, e armazenamos alguns valor nelas.

Adiante podemos resgatar o conteúdo dessas variáveis e exibi-los na linha de comando do print.

Para juntar o valor da variável a um String, utilize o operador **\(...)** .

Dentro dos parênteses, deve ir o nome da variável que você está tentando acessar.

Resultado:

```
Nome: Felpudo Jr. - Idade: 12 - Altura: 0.55
```

9. Operações com Strings

O tipo **String** se refere a uma palavra, frase ou qualquer cadeia de vários caracteres (ou letras).

Existem algumas operações (ou métodos) que podemos utilizar com este tipo de objeto. Confira alguns:

a) uppercased()

Converte todos os caracteres em maiúsculo.

b) count

Nos informa o tamanho do String, ou quantas letras (caracteres) ele tem.

c) replacingOccurrences(of: a, with: b)

Substitui o caractere do argumento 'a' (parâmetro antes da vírgula) pelo argumento 'b' (depois da vírgula).

*Observação: Para os métodos com String funcionarem, precisaremos carregar mais uma biblioteca de comandos do Swift, o **UIKit**. Para isso usamos o comando **import UIKit** logo no início do código.

Veja no quadro de exemplo esses métodos funcionando:

Exemplo:

```
import UIKit

var nome = "Felpudo Júnior"

print(nome.count)
print(nome.uppercased())
print(nome.lowercased())
print(nome.replacingOccurrences(of: "Fel", with: "ABCD"))
```

Resultado:

```
14
FELPUDO JÚNIOR
felpudo júnior
ABCDpudo Júnior
```

9. Comandos Unicode para Strings

Ao utilizar um **String** podemos usar operadores **Unicode** (Padrão Internacional de Caracteres), que nos auxiliam a montar e formatar textos para exibir mensagens no console, por exemplo.

COMANDO UNICODE	DESCRIÇÃO
\t	Insere um parágrafo
\n	Quebra de Linha
\"	Aspas Duplas
\'	Aspas SImples
****	Barra Invertida

Execute no seu compilador alguns comandos Unicode para testar a formatação dos Strings.

Exemplo:

```
print("Eu sou o Felpudo!")
print("\tEu sou o Felpudo!")
print("\nEu sou o Felpudo!")
print("\"Eu sou o Felpudo!\"")
print("\'Eu sou o Felpudo!\'")
print("\\Eu sou o Felpudo!\\")
```

Resultado:

```
Eu sou o Felpudo!
	Eu sou o Felpudo!

Eu sou o Felpudo!
"Eu sou o Felpudo!"
'Eu sou o Felpudo!'
\Eu sou o Felpudo!\
```

Perceba, nos últimos três exemplos, que a **barra invertida** (****) dentro de uma cadeia de caracteres faz o compilador **ignorar o próximo caractere** e com isso conseguimos inserir uma barra invertida ou aspas dentro do **String**.

10. Convertendo Tipos de Variáveis

Em determinados momentos na programação, será necessário manipular os **Tipos das Variáveis** para chegarmos em determinados resultados.

Adiante, neste material, iremos nos deparar com situações como esta. Por isso vamos entender primeiros as operações para **Converter os Tipos de Variáveis**.

Veja alguns exemplos de conversão de tipos:

a) **String** para **Int**

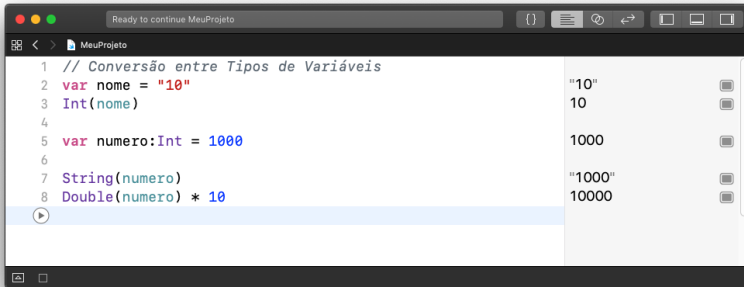
Exemplo:

```
var palavra = "500"
var numero = Int(palavra)
```

b) **Int** para **String**

Exemplo:

```
var numero = 10
var palavra = String(numero)
```

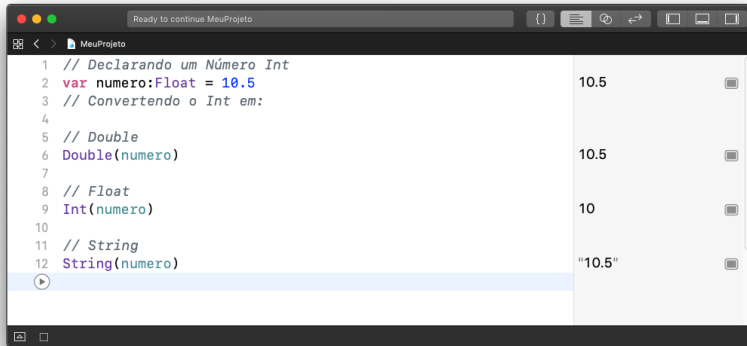
Teste alguns destes comandos no Playground.

Perceba que no Console Lateral, podemos enxergar os dados como String (presença das aspas duplas) ou como números (sem aspas).

Para converter os tipos no Swift, basta utilizar o nome do tipo, e passar o argumento que deseja converter entre parênteses.

Exemplo:

```
// Declaração de um String
var nome = "10"

// Convertendo o String em Int
Int(nome) *10
```

Mais adiante utilizaremos estas conversões, veremos que em muitos casos, a conversão de tipos será necessária.

Por enquanto saiba que podemos converter os tipos facilmente usando este padrão:

Exemplo:

```
// Declarando um Float
var numero:Float = 10.5

// Convertendo o Float em:
// Double
Double(numero)

// Float
Int(numero)

// String
String(numero)
```

11. Operações Matemáticas

Um recurso indispensável para a programação é o de fazer facilmente **Operações Matemáticas** pelo Compilador.

As **Quatro Operações Fundamentais** são realizadas facilmente pelos operadores:

+ Soma

- Subtração

***** Multiplicação

/ Divisão

Podemos executar tanto as **Operações Fundamentais da Aritmética**, quanto trabalhar com expressões e

funções trigonométricas mais avançadas.

a) Módulo

O módulo é calculado com o operador **%**. Ele nos retorna o **resto** entre a divisão de dois números.

Exemplo:

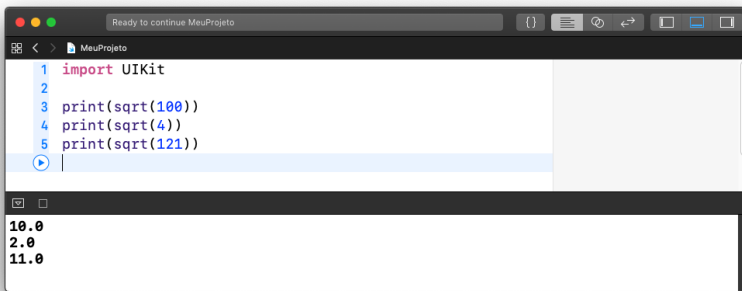
```
print( 7%3 )
print( 8%2 )
print( 4%5 )
```

Resultado:

```
1
0
4
```

b) Raiz Quadrada - sqrt()

O comando **Math** nos permite realizar operações ainda mais avançadas como o **cálculo da raiz quadrada**.

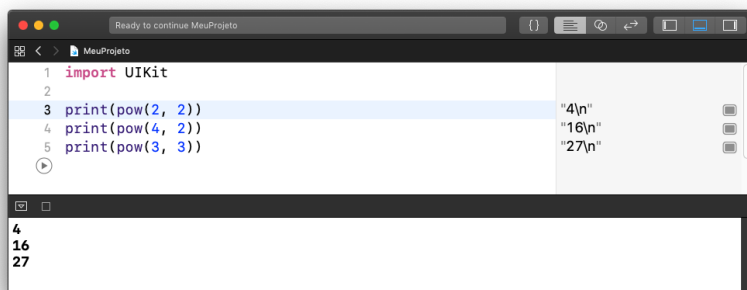
Lembre-se que para usar as funções matemáticas, você precisará importar a classe UIKit. Veja o exemplo:

Exemplo:

```
import UIKit

print(sqrt(100))
print(sqrt(4))
print(sqrt(121))
```

Resultado:

```
10.0
2.0
11.0
```

c) Potência - pow()

O **pow** também nos permite elevar à **potência** de um número.

Exemplo:

```
print( pow(2, 2) )
print( pow(4, 2) )
print( pow(3, 3) )
```

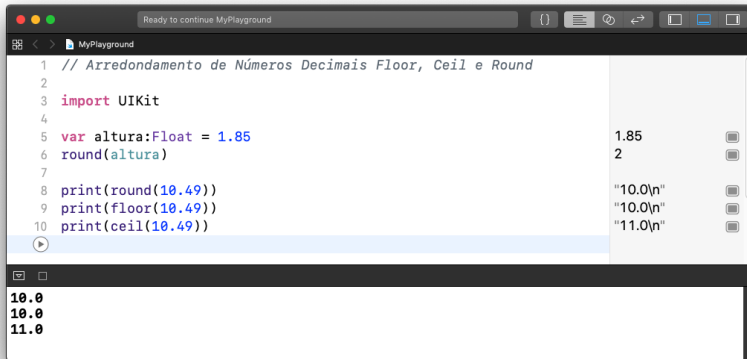
Resultado:

```
4
16
27
```

12. Arredondamento de Decimais

Conheça também alguns métodos para arredondar números decimais.

```
1  // Arredondamento de Números Decimais Floor, Ceil e Round
2
3  import UIKit
4
5  var altura:Float = 1.85                 1.85
6  round(altura)                           2
7
8  print(round(10.49))                     "10.0\n"
9  print(floor(10.49))                     "10.0\n"
10 print(ceil(10.49))                      "11.0\n"
```

```
10.0
10.0
11.0
```

```
import UIKit

var altura:Float = 1.85
round(altura)

print(round(10.49))
print(floor(10.49))
print(ceil(10.49))
```

a) round()

Retorna o **número inteiro mais próximo** do número decimal indicado.

Exemplo:

```
print(round(3.49) )
print(round(3.1) )
print(round(3.501) )
```

Resultado:

```
3
3
4
```

b) floor()

Retorna o **número inteiro abaixo** do decimal indicado.

Exemplo:

```
print( floor(7.99) )
print( floor(7.01) )
```

Resultado:

```
7
7
```

c) ceil()

Retorna o **próximo número inteiro acima** do decimal apontado.

Exemplo:

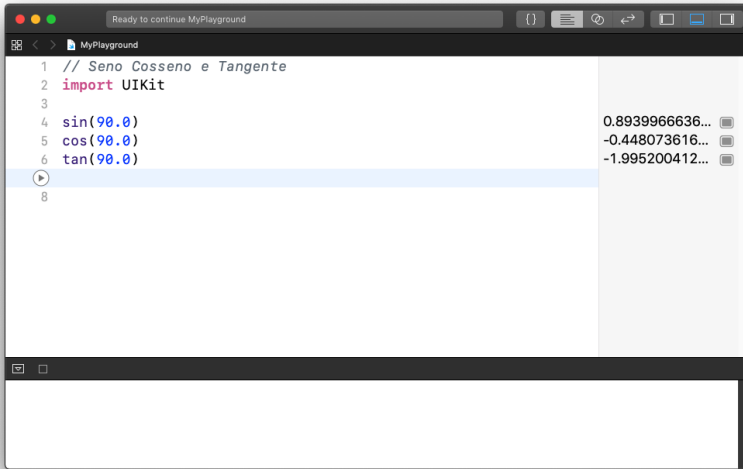
```
print( ceil(7.01) )
print( ceil(7.99) )
```

Resultado:

```
8
8
```

14. Ângulos e Pi Radianos

```
// Ângulos e PI Radianos
var angulo:Double = 360
var radianos:Double = Double.pi

// Converter Ângulo em Radianos
print(angulo * (Double.pi/180))

// Converter Radianos em Ângulos
print(radianos * (180/Double.pi))
```

15. Seno Cosseno e Tangente

```
// Seno Cosseno e Tangente
import UIKit

sin(90.0)
cos(90.0)
tan(90.0)
```

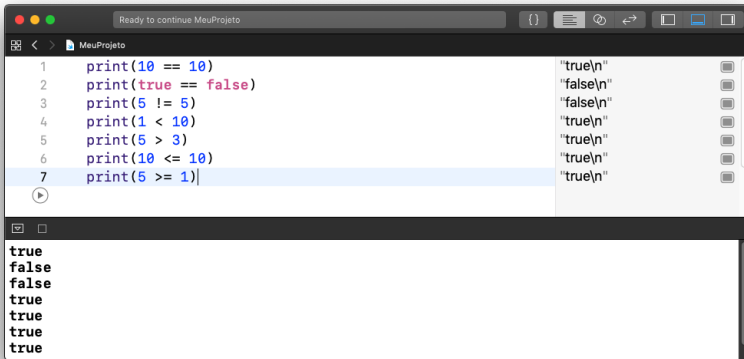
16. Operadores de Comparação

Podemos comparar dois valores e obter **true** ou **false** caso a comparação seja satisfeita ou não.

Observe quais são os **Operadores de Comparação**:

OPERADOR	COMPARAÇÃO
==	igual
!=	diferente
<	menor
>	maior
<=	menor ou igual
>=	maior ou igual

Faça alguns testes no Playground e observe os resultados.

```
Ready to continue MeuProjeto
MeuProjeto
1  print(10 == 10)          "true\n"
2  print(true == false)     "false\n"
3  print(5 != 5)            "false\n"
4  print(1 < 10)            "true\n"
5  print(5 > 3)             "true\n"
6  print(10 <= 10)          "true\n"
7  print(5 >= 1)            "true\n"

true
false
false
true
true
true
true
```

Exemplo:

```
print(10 == 10)
print(true == false)
print(5 != 5)
print(1 < 10)
print(5 > 3)
print(10 <= 10)
print(5 >= 1)
```

Resultado:

```
true
false
false
true
true
true
true
```

14. Operadores de Atribuição

Com estes operadores, você pode abreviar um pouco a escrita do código.

Por exemplo, para incrementar o valor de uma variável, podemos realizar o seguinte comando:

Exemplo:

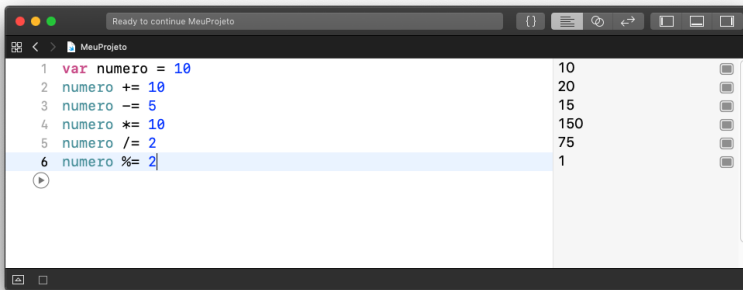
```
var numero = 10
numero = numero + 10
```

Resultado:

```
20
```

Ou seja, iremos armazenar na variável número, o valor da própria variável somado a 10.

Este comando pode ser abreviado se escrito da seguinte forma:

Exemplo:

```
var numero = 10
numero += 10
```

Resultado:

```
20
```

Seguindo este padrão, podemos utilizar os operadores aritméticos de soma, subtração, multiplicação, divisão e módulo para realizar esta operação.

Veja no quadro abaixo, um teste de valores utilizando os **Operadores de Atribuição**:

15. Números Aleatórios (Randômicos)

Os números aleatórios (*Random*) são utilizados constantemente.

Tente imaginar onde eventos aleatórios podem estar acontecendo em aplicações que utilizamos no dia a dia:

- Nos ***games***, para realizar variações de eventos e jogadas
- Na ordenação das postagens que vemos na *timeline* do *Facebook* por exemplo
- Na segurança de informação para a geração de senhas e identificações

Para criar um número aleatório em Swift, utilizamos a seguinte sintaxe:

Exemplo:

```
print(Int.random(in:0..<5))
```

Este comando vai gerar um número inteiro aleatório entre 0 e 4 (menor que 5)
Chamamos de operador de intervalo aberto

Também podemos utilizar a seguinte sintaxe para obter o mesmo resultado

Exemplo:

```
print(Int.random(in:0...4))
```

Neste caso, o compilador vai nos gerar um número aleatório de 0 até 4

Podemos também gerar números aleatórios com outros tipos de variáveis:

Exemplo:

```
print(Float.random(in:0..<10.0))
print(Bool.random())
```

No exemplo acima, estamos gerando valores aleatórios para dados do tipo **Float** e **Boolean**.

16. Variáveis Opcionais

Em Swift, podemos trabalhar com variáveis vazias, que chamamos de Opcionais. ‘

17. Operadores Lógicos

As operações lógicas servem para comparar dois valores *Booleanos* e retornar um resultado referente à esta comparação.

Na programação existem basicamente três Operadores Lógicos:

OPERADOR	NOME	TRADUÇÃO
&&	And	E
||	Or	OU
!	Not	NÃO

a) Operador And (&&)

Compara dois *Booleanos* e retornar **true** apenas se o primeiro valor for **true E** o segundo valor também for **true**.

Exemplo:

```
print(true && true)
print(true && false)
print(false && true)
print(false && false)
```

Resultado:

```
true
false
false
false
```

b) Operador Or (||)

Vai comparar dois *Booleanos* e retornar **true** se o primeiro valor for **true OU** o segundo valor for **true**.

Exemplo:

```
print(true || true)
print(true || false)
print(false || true)
print(false || false)
```

Resultado:

```
true
true
true
false
```

c) Operador Not (!)

Inverte o valor do *Booleano*.

Exemplo:

```
print( ! true )
print( ! false )
```

Resultado:

```
false
true
```

18. Condição If

A **Condição If** é utilizada para controlar o fluxo da aplicação digital. É ela quem diz por qual caminho a aplicação deve seguir sua execução, baseada em uma verificação.

A **Condição If** (Se) faz a execução do programa seguir determinado caminho, dependendo do resultado de um teste lógico.

Se o teste resulta em **true** (verdadeiro) o programa segue executando um determinado trecho de código.

Caso o teste resulta em **false** (falso) o programa ignora este trecho de código ou, até mesmo, executa outro determinado trecho.

Veja uma implementação da estrutura de Condição If / Else no Swift.

```
var idade = 16

if ( idade >= 18 ) {
    print("Pode Dirigir!")
} else {
    print("Nao Pode Dirigir!")
}
```

```
Nao Pode Dirigir!
```

Exemplo:

```
var idade = 16
```

```
if ( idade >= 18 ) {
        print("Pode Dirigir!")
} else {
        print("Nao Pode Dirigir!")
}
```

Caso a comparação, que está entre parênteses (idade >= 18), seja satisfeita o programa vai executar o que está dentro das primeiras chaves **{...}**.

Exemplo:

```
if ( idade >= 18 ) {
        print("Pode Dirigir!")
} else {
        print("Nao Pode Dirigir!")
}
```

Caso o resultado da comparação seja **false,** o programa ignora o que está contido nas primeiras chaves e executa o que está na segunda, depois do **else**.

Exemplo:

```
if ( idade >= 18 ) {
        print("Pode Dirigir!")
} else {
        print("Não Pode Dirigir!")
}
```

O **else** é opcional! Podemos criar uma condição **if** sem o **else**. Caso a condição seja falsa e o **else** não exista, o programa ignora tudo e continua a execução.

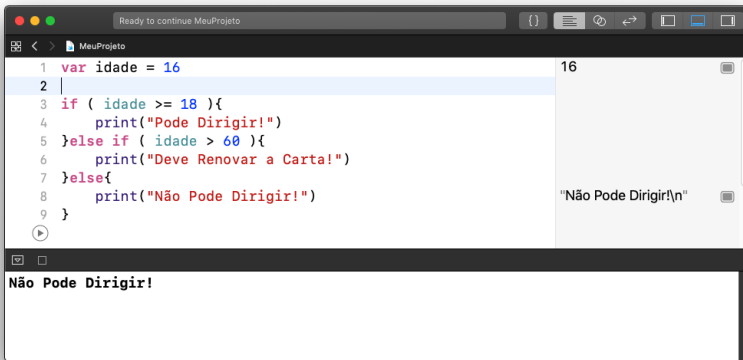
Existe também a estrutura **else if**, onde podemos utilizar uma segunda comparação antes de executar o que existe dentro das chaves do **else.**

Exemplo:

```
var idade = 16

if ( idade >= 18 ){
    print("Pode Dirigir!")
}else if ( idade > 60 ){
    print("Deve Renovar a Carta!")
}else{
    print("Não Pode Dirigir!")
}
```

Baseado neste exemplo, perceba que você pode utilizar quantos **if/else** quiser e comparar diferentes valores e situações.

```swift
var idade = 16

if ( idade >= 18 ){
    print("Pode Dirigir!")
}else if ( idade > 60 ){
    print("Deve Renovar a Carta!")
}else{
    print("Não Pode Dirigir!")
}
```

```
Não Pode Dirigir!
```

Observe que os conceitos de **Comparações Lógicas** e **Estruturas de Condição** são utilizados no desenvolvimento de qualquer tipo de Aplicação Digital, como em Dispositivos Eletrônicos, Robótica e *Videogames*.

A **Estrutura de Condição** é o que faz parecer que o computador **Pensa** e toma suas **Próprias Decisões**.

19. Escopos ou Limites

A palavra **Escopo** trata-se de definir um **limite** de uma **Ação**, com **começo e fim.** Já falamos sobre isso brevemente, mas agora é hora de entender um pouco mais.

Trata-se de um conceito delimitante, onde podemos trabalhar com **Valores** e **Objetos**.

Entenda que tudo que está dentro das chaves **{...}** faz parte do mesmo contexto.

Ao usar uma condição **if** estamos criando um novo **limite** na nossa programação. Imagine como o conteúdo contido dentro de outro trecho de código.

Exemplo:

```
var idade = 21

if ( idade > 18 ) {
    print("Tem \(idade) anos. Pode Dirigir!")
}
```

A estrutura das chaves também representa uma **hierarquia** (conceito de parentesco, ou pai e filho).

Perceba que a estrutura de condição **if** está contida dentro do código principal.

Como a variável idade, foi declarada no pai, ela pode ser acessada pelo filho (dentro dos limites do **if**).

Observe atentamente o exemplo a seguir:

Exemplo:

```
var idade = 16

if ( idade >= 18 ){
    var mensagem = "Pode Dirigir!"
}

print(mensagem)
```

Se você tentar executar este código, ele vai **não vai funcionar**. Perceba que a variável está sendo criada dentro da estrutura **if**, e está tentando ser acessada de fora do limite.

Diferente do exemplo a seguir, onde a variável foi criada no pai (fora do limite do **if)**, e acessada no filho (estrutura **if**).

Exemplo:

```
var idade = 16
var mensagem = “ ”

if ( idade >= 18 ){
   mensagem = "Pode Dirigir!"
}

print(mensagem)
```

No exemplo, fizemos apenas uma mudança sutil, e declaramos a variável **mensagem** fora do **if**. Isto faz com que a variável seja visível em por todo os escopos ou limites do código.

Lembre-se que sempre que há a presença das chaves **{...}** estamos falando de um contexto dentro do outro.

Lembre-se também que todo limite deve ter **início** e **fim**. Ou seja, sempre que **abrir a chave**, devemos **fechá-la**!

20. Projeto #1 - Jogo do Par ou Ímpar

Agora, utilizaremos todo conhecimento adquirido até aqui. Criaremos um **Jogo de Par ou Ímpar**. Antes de partir para a programação, é importante que você tenha o problema 100% solucionado na sua cabeça.

Para isso, recomendo que você use o bom e velho lápis e papel para você desenhar esse **Algoritmo** à mão, até visualizar muito bem como ele vai funcionar. Só então comece a resolver as **Linhas de Programação**.

Pense como poderia ser desenvolvido um **Algoritmo** para resolver este problema. Tente dividir todo o processo em etapas. Por exemplo:

- Sortear dois Números Aleatórios entre 0 e 5
- Exibir os Valores das jogadas sorteadas
- Verificar se a soma das jogadas é par ou é ímpar
- Exibir mensagem dizendo quem venceu

Se você absorveu o conteúdo deste Livro passo a passo até aqui, já tem todos os conhecimentos necessários para construir este **Algoritmo**.

Lembre-se que podem existir diferentes maneiras de se escrever um programa ou algoritmo de computador. O importante é chegarmos ao objetivo traçado.

Deixarei o problema aqui como um desafio pra você resolver.

DESAFIO #3

Eu te desafio a utilizar tudo o que você aprendeu até aqui, e criar um Jogo de Par ou Ímpar utilizando a Linguagem Swift.

Adiante você tem uma resolução, caso precise de ajuda, mas o mais importante é tentar e errar. Assim que se aprende de verdade!

Lesmo

Tente se esforçar e resolver o desafio antes de olhar a minha resolução.

Uruca

DESAFIO #4

Jogue o jogo que você programou com um amigo, diga à ele que foi você mesmo quem criou. Que escreveu este programa, linha por linha de código.

Depois ensine seu amigo a programar. Lembre-se que ensinar também é uma ótima maneira de aprender.

A seguir você pode conferir o meu **Código final comentado** com a minha resolução do problema.

Lembre-se que existem diversas maneiras diferentes para desenvolver este algoritmo.

O mais importante é que no final você alcance um resultado satisfatório na sua aplicação, utilize o mínimo possível de código, e divirta-se bastante no desenvolvimento.

Boa Diversão e Bons Estudos!

Ready to continue MyPlayground

MyPlayground

```
1  // Jogo do Par ou Ímpar
2  // Jogador A - PAR
3  // Jogador B - ÍMPAR
4
5  var jogadaA = Int.random(in: 0...5)                    5
6  var jogadaB = Int.random(in: 0...5)                    4
7
8  print("JogadaA \(jogadaA)")                           "JogadaA 5\n"
9  print("JogadaB \(jogadaB)")                           "JogadaB 4\n"
10
11 if((jogadaA+jogadaB)%2 == 0){
12     print("Número PAR - Jogador A Venceu!")
13 }else {
14     print("Número ÍMPAR - Jogador B Venceu!")        "Número ÍMPAR - J...
15 }
```

```
JogadaA 5
JogadaB 4
Número ÍMPAR - Jogador B Venceu!
```

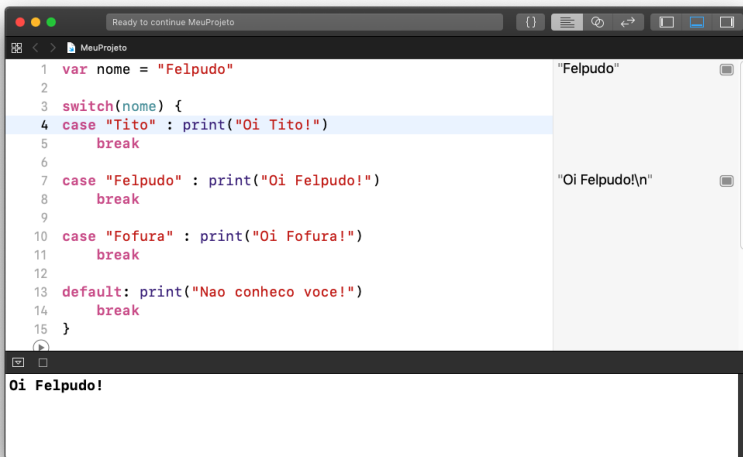
Resolução do Projeto #1 - Jogo do Par ou Ímpar

```
// Jogo do Par ou Ímpar
// Jogador A - PAR
// Jogador B - ÍMPAR

var jogadaA = Int.random(in: 0...5)
var jogadaB = Int.random(in: 0...5)

print("JogadaA \(jogadaA)")
print("JogadaB \(jogadaB)")

if((jogadaA+jogadaB)%2 == 0){
    print("Número PAR - Jogador A Venceu!")
}else {
    print("Número ÍMPAR - Jogador B Venceu!")
}
```

21. Estrutura de Condição Switch

Em alguns casos, ao invés de utilizar vários **if, else if…** podemos simplificar o **Código** utilizando a estrutura **switch**. Esta estrutura compara um **Valor** a várias outras possibilidades, caso a comparação seja verdadeira, o **Comando** determinado é executado.

```
var nome = "Felpudo"

switch(nome) {
case "Tito" : print("Oi Tito!")
    break

case "Felpudo" : print("Oi Felpudo!")
    break

case "Fofura" : print("Oi Fofura!")
    break

default: print("Nao conheco voce!")
    break
}
```

```
Oi Felpudo!
```

Exemplo:

```
String nome = "Felpudo";

switch(nome) {
    case "Tito" : print("Oi Tito!") break;
    case "Felpudo" : print("Oi Felpudo!") break;
    case "Fofura" : print("Oi Fofura!") break;
    default: print("Nao conheco voce!") break;
}
```

Perceba no **Código** do exemplo, os **Comandos** destacados em negrito fazem parte da estrutura do **switch**.

Entenda cada um dos operadores da estrutura **Switch**:

- **switch()** seleciona a **Variável** a ser comparada.
- **case** compara com o valor do próximo argumento.
- **:** Indica o início do comando a ser executado.
- **default** será executado, caso nenhuma das comparações anteriores tenha sido satisfeita.
- **break** - Indica o fim da ação executada. Faz com que a execução pule para fora do escopo na execução do código.

Adiante veremos que o **break** pode ser utilizado também em outras estruturas na programação para cessar a execução do escopo atual.

22. Projeto #2 - Jo Ken Pô

Já sabemos o que é necessário para desenvolver este jogo do **Pedra, Papel ou Tesoura**. Perceba que a mecânica, a lógica do jogo, é bem similar ao Jogo do Par ou Ímpar. Desafio você a criar sozinho este jogo.

Tenho certeza que neste desenvolvimento muitos conceitos e idéias serão formadas no seu raciocínio. Isto é o mais importante.

Lembrando que existem diversas maneiras diferentes de se resolver um problema como este.

Uruca

DESAFIO #5

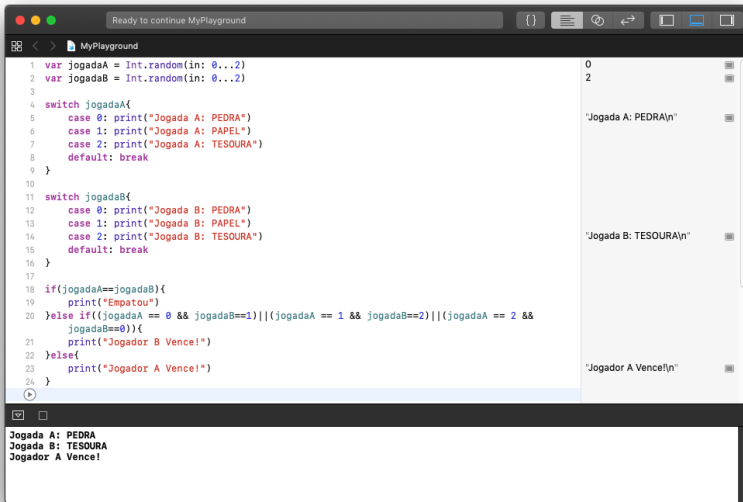
Crie já seu Algoritmo em *Swift* do Jogo Jokenpô! Uma dica, as estruturas switch ajudam a diminuir o uso da estrutura if/else.

Mãos à obra, ao final compare seu algoritmo com a minha solução que está adiante.

DESAFIO #6

Desenvolva novamente o algoritmo do Jokenpô, porém desta vez utilizando uma lógica totalmente diferente do primeiro algoritmo.

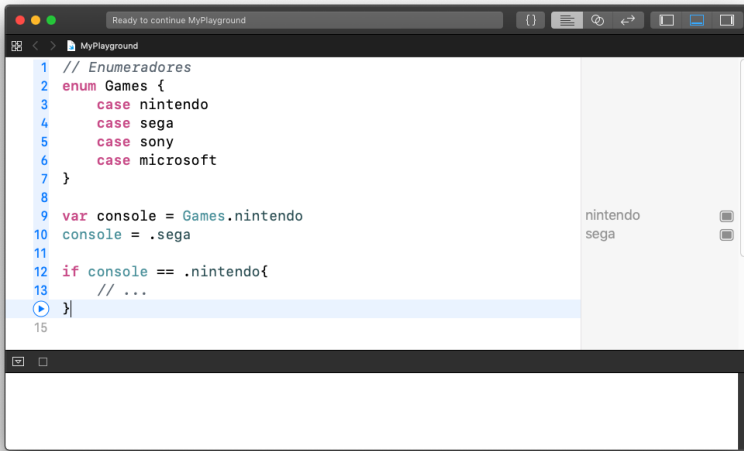
Compartilhe sua solução na sua rede social e marque o professor Tito Petri.

Lesmo

Resolução do Projeto #2 - Jo Ken Pô

```
var jogadaA = Int.random(in: 0...2)
var jogadaB = Int.random(in: 0...2)

switch jogadaA{
    case 0: print("Jogada A: PEDRA")
    case 1: print("Jogada A: PAPEL")
    case 2: print("Jogada A: TESOURA")
    default: break
}

switch jogadaB{
    case 0: print("Jogada B: PEDRA")
    case 1: print("Jogada B: PAPEL")
    case 2: print("Jogada B: TESOURA")
    default: break
}
```

```
if(jogadaA==jogadaB){
    print("Empatou")
}else if((jogadaA == 0 && jogadaB==1)||(jogadaA == 1 &&
jogadaB==2)||(jogadaA == 2 && jogadaB==0)){
    print("Jogador B Vence!")
}else{
    print("Jogador A Vence!")
}
```

23. Enumeradores

```
// Enumeradores
enum Games {
    case nintendo
    case sega
    case sony
    case microsoft
}

var console = Games.nintendo
console = .sega

if console == .nintendo{
    // ...
}
```

```
// Declarando a Estrutura do Enumerador

enum Jogada {
    case pedra
    case papel
    case tesoura
}

// Inicializando as variáveis do tipo Enumerador

var jogadaA = Jogada.pedra
jogadaA = .pedra
```

```
// Acessando e Comparando um Enumerador

if jogadaA == .tesoura{
    // ...
}
```

```
// Enumeradores
enum Jogada{
    case pedra
    case papel
    case tesoura
}

var jogadaA:Jogada = Jogada.pedra
var jogadaB:Jogada = Jogada.pedra

switch(Int.random(in: 0...2)){
case 0: jogadaA = Jogada.pedra; break
case 1: jogadaA = Jogada.papel; break
case 2: jogadaA = Jogada.tesoura; break
default: break
}

switch(Int.random(in: 0...2)){
case 0: jogadaB = Jogada.pedra; break
case 1: jogadaB = Jogada.papel; break
case 2: jogadaB = Jogada.tesoura; break
default: break
}

print(jogadaA)
print(jogadaB)

if(jogadaA == jogadaB){
    print("Empatou")
}else if( ((jogadaA == .pedra) && (jogadaB == .tesoura)) ||
    ( (jogadaA == .papel) && (jogadaB == .pedra)) || (
    (jogadaA == .tesoura) && (jogadaB == .papel) )) {
    print("Jogador A Vence!")
}else{
    print("Jogador B Vence!")
}
```

Exemplo:

```
// Enumeradores
enum Jogada{
    case pedra
    case papel
    case tesoura
}

var jogadaA:Jogada = Jogada.pedra
var jogadaB:Jogada = Jogada.pedra

switch(Int.random(in: 0...2)){
case 0: jogadaA = Jogada.pedra; break
case 1: jogadaA = Jogada.papel; break
case 2: jogadaA = Jogada.tesoura; break
default: break
}

switch(Int.random(in: 0...2)){
case 0: jogadaB = Jogada.pedra; break
case 1: jogadaB = Jogada.papel; break
case 2: jogadaB = Jogada.tesoura; break
default: break
}

print(jogadaA)
print(jogadaB)

if(jogadaA == jogadaB){
    print("Empatou")
}else if( ((jogadaA == .pedra) && (jogadaB == .tesoura)) || (
(jogadaA == .papel) && (jogadaB == .pedra)) || ( (jogadaA ==
.tesoura) && (jogadaB == .papel) )) {
    print("Jogador A Vence!")
}else{
    print("Jogador B Vence!")
}
```

23. Repetição for

A estrutura de repetição **for** serve para executar um determinado comando por várias vezes.

Podemos determinar o número de vezes que uma instrução será executada.

Execute a estrutura **for** abaixo, e observe o resultado no console.

```
Ready to continue MeuProjeto
MeuProjeto
1  for _ in 1...5{
2      print("Bem-Vindo ao Swift")
3  }
4
"Bem-Vindo ao Swift\...

Bem-Vindo ao Swift
Bem-Vindo ao Swift
Bem-Vindo ao Swift
Bem-Vindo ao Swift
Bem-Vindo ao Swift
```

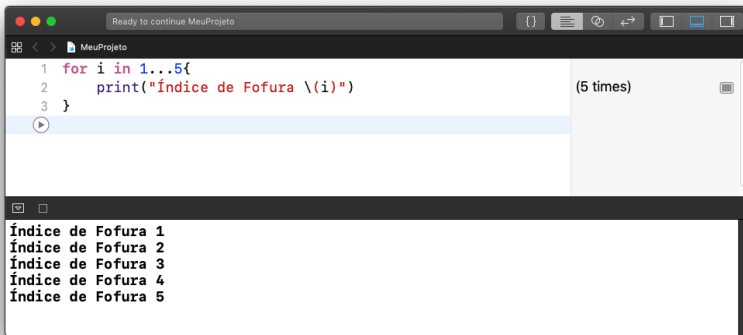
Exemplo:

```
for _ in 1...5{
    print("Bem-Vindo ao Swift")
}
```

Perceba que o comando **print** foi executado 5 vezes.

Perceba que usamos o *underline* _ na criação da estrutura **for**. Isso caracteriza que não iremos utilizar nenhuma variável para contar o número de execuções do **for**.

Agora tente executar este outro exemplo:

```
Ready to continue MeuProjeto
MeuProjeto
1  for i in 1...5{
2      print("Índice de Fofura \(i)")
3  }
(5 times)

Índice de Fofura 1
Índice de Fofura 2
Índice de Fofura 3
Índice de Fofura 4
Índice de Fofura 5
```

Exemplo:

```
for i in 1...5{
    print("Índice de Fofura \(i)")
}
```

Perceba que agora ao invés do *underline* _ declaramos uma variável (geralmente chamada de **i** por boas normas de programação) durante a execução, a variável numérica chamada de **i** foi incrementada (aumentou de valor) a cada vez que o comando foi executado.

Agora vamos entender o que aconteceu. Observe como foi criada a estrutura do **for**.

Exemplo:

```
for i in 1...5{
    print("Índice de Fofura \(i)")
}
```

É muito importante entender cada um destes argumentos para utilizar o **for**.

O **primeiro argumento (i)** cria uma variável do tipo inteira e chama de **i**. Inicializamos este número por 1 e ele serve como um **contador** e nos indicar quantas vezes a estrutura já foi executada.

** Utilizamos a letra **i** por questões de boas práticas de programação. Este **contador** poderia ser declarado por qualquer nome.*

O **segundo argumento (in)** especifica o intervalo de valores que o contador **(i)** vai assumir. No exemplo, mandamos o contador iniciar em 1 e ir até 5.

Ou seja, durante a primeira execução do laço, a variável **i** terá o valor **1**. Durante a segunda, o valor **2**, durante a terceira o valor **3**, e assim sucessivamente.

Veja a seguir mais algumas implementações da estrutura (ou laço) de repetição **for**:

a) Decrementando o Contador - reversed()

Podemos executar a estrutura for de trás para frente com o comando **reversed()**

Execute os comando a seguir no seu Playground e observe os resultados:

Exemplo:

```
for i in (1...10).reversed(){
    print("Índice de Fofura \(i)")
}
```

b) Incrementando o Contador por Intervalos

Também podemos incrementar ou decrementar o valor do contador de acordo com um número específico (de 2 em 2, de 3 em 3). Observe o exemplo:

Exemplo:

```
for i in stride(from: 0, to: 30, by: 3){
    print("Índice de Fofura \(i)")
}
```

No caso acima, o valor do contador vai de 0 a 30, em intervalos de 3 em 3.

Também podemos inverter a execução do **for** e especificando o intervalo utilizando a seguinte sintaxe:

Exemplo:

```
for i in stride(from: 30, to: 0, by: -2){
    print("Índice de Fofura \(i)")
}
```

Neste caso o **for** vai de 30 até 0, decrementando seu valor em intervalos de -2 em -2.

DESAFIO #7

Crie um algoritmo que imprima no console todos os números pares de 0 a 50.

Lesmo

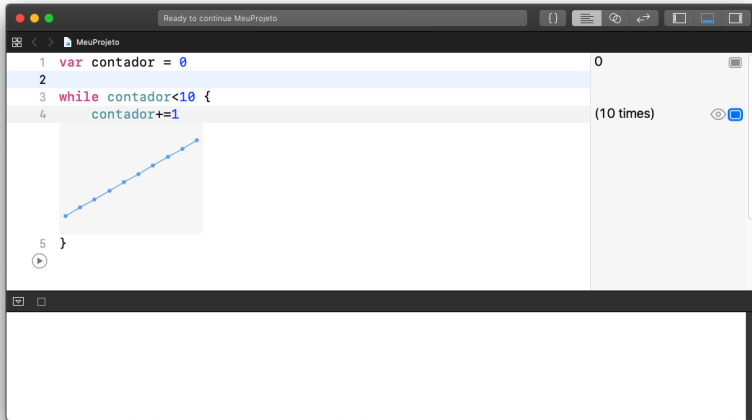
Bugado

DESAFIO #8

Agora crie um algoritmo que imprime no console todos os números divisíveis por 5 no intervalo de 0 a 100.

Compartilhe esta solução com seus amigos na sua rede social e marque o professor Tito Petri.

24. Repetição while

A estrutura **while** é bem parecida com o **for**, porém a sua implementação é um pouco diferente. Execute o seguinte **Código** abaixo e observe o resultado obtido.

Exemplo:

```
var contador = 0

while contador<10 {
    contador+=1
}
```

Para utilizar a estrutura **while**, precisa existir **três argumentos**, que são criados e manipulados separadamente. Contador, Condição e Incremento

Exemplo:

```
var contador = 0 // contador

while contador<10 { // condição
   contador += 1 // incremento
}
```

Preste muita atenção ao utilizar a estrutura **while**. Se você esquecer de modificar o **Valor** do **Contador**, a fim de quebrar a condição do **while**, isso fará com que a execução do programa **nunca saia do *loop***. Isto é o que chamamos de ***loop* infinito**.

Existem poucas diferenças entre se utilizar uma estrutura **for** e **while**.

A estrutura **for** sempre vai exigir que o **contador** seja uma **nova variável**, declarado na criação do **for**.

Já a estrutura **while** pode utilizar uma **variável já existente** no código ou no projeto.

O **while** também nos dá mais flexibilidade para manipular o **contador** em situações mais específicas.

25. Arrays, Listas ou Vetores

São estruturas de dados ordenadas que armazena vários objetos (variáveis) do mesmo tipo.

Quando você ouvir os termos Lista**,** Vetor**,** Array**,** Matriz**,** perceba que se tratam basicamente do mesmo conceito.

Ao invés de trabalhar com um único valor ou variável, podemos utilizar uma estrutura ordenada onde vários objetos ficam arrumados e são acessados por coordenadas.

Veja o exemplo a seguir. Ao invés de armazenar um único valor (palavra) dentro de um String, podemos criar uma lista, e armazenar diversas palavras, separando-as por índices, ou posições dentro desta lista.

Observe o código a seguir. Ele exemplifica como podemos criar uma lista de palavras (Strings).

Exemplo:

```
var nomes:[String] = ["Felpudo", "Fofura", "Lesmo", "Bugado",
"Peluche", "Uruca", "Felpudo Jr."]
```

Perceba que, ao declarar a variável, utilizamos o **tipo** (String) entre **colchetes [ ]**.

Os **colchetes** definem que a variável criada será um **Array de Strings**.

Depois, definimos o nome da variável (*array)* como **lista** ou qualquer outro nome.

Para inicializar a **lista** com algum valor precisamos utilizar um array de objetos.

Este array é definido por **colchetes [...]** e irá conter vários objetos do tipo String, separados por uma vírgula entre eles.

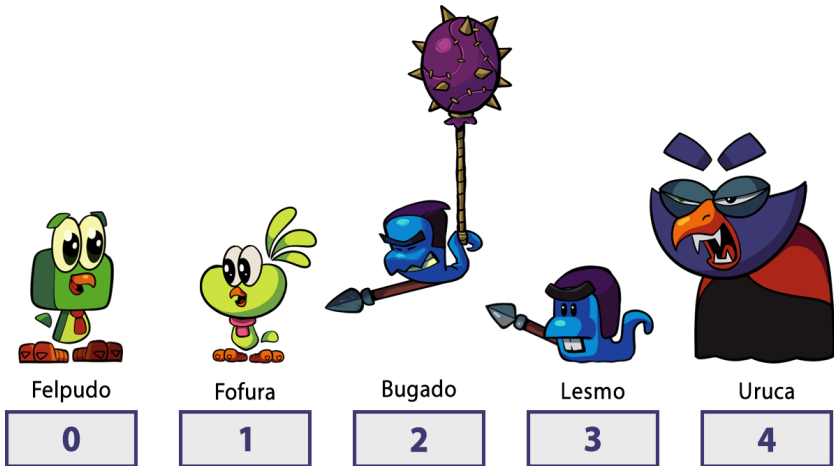
Veja, a seguir, a declaração de arrays de outros tipos como **Boolean**, **int,** e **Float**.

Exemplo:

```
var numeros:[Int] = [1,100,30,5]
var estados:[Bool] = [true,true,true,false]
var decimais:[Float] = [1.001, 5.5, 40.1, 99.9]
```

Então, lembre-se que para definir uma variável como uma lista, basta adicionar **[]** ao tipo, na hora da declaração.

Já armazenamos uma listagem de dados dentro destas variáveis, agora precisamos **acessar** estes dados.

Para fazer a **leitura do dado** de uma lista, devemos utilizar o índice da lista.

O índice é a posição de cada objeto dentro do array.

Lembre-se que a primeira posição é a casa 0.

Exemplo:

```
var nomes:[String] = ["Felpudo", "Fofura", "Lesmo", "Bugado",
"Peluche", "Uruca", "Felpudo Jr."]
```

"Felpudo" - ocupa o **índice 0** da lista
"Fofura" - ocupa o **índice 1** da lista
"Lesmo" - ocupa o **índice 2** da lista
e assim sucessivamente…

Para acessar um índice da lista **nomes** utilizamos o nome da variável e o número do índice entre **colchetes**.

nomes[0] - vai nos trazer o valor contido no primeiro índice (ou posição) da lista.

Exemplo:

```
var nomes:[String] = ["Felpudo", "Fofura", "Lesmo", "Bugado",
"Uruca"]

print(nomes[0])
print(nomes[1])
print(nomes[4])
```

Resultado:

```
Felpudo
Fofura
Uruca
```

Perceba que a lista tem **cinco itens**, porém o índice da **última casa é 4**, justamente porque o **índice da primeira casa é 0**.

26. Array Multidimensional ou Matriz

Além das listas, lembre-se que existem também as Matrizes de Dados, ou Arrays Multidimensionais.

As Matrizes utilizam o conceito de organização em grade. Seria como acessar um objeto referenciando o índice da **linha** e da **coluna.**

Exemplo:

```
var cidades:[[String]] = [ ["São Paulo", "Rio de Janeiro", "Belo Horizonte", "Espírito Santo"], ["Nova Iorque", "São Francisco", "Los Angeles", "Orlando"], ["Patópolis", "Cucamonga", "Springfield", "Acapulco"]]
```

Perceba que agora a **Matriz** foi declarada utilizando-se **2 colchetes**. E inicializada com um "array de arrays", ou seja, um array com 3 arrays dentro, separados por vírgula, sempre dentro das chaves.

Para acessar algum dado nesta **matriz**, você deve especificar as duas coordenadas dos índices.

Exemplo:

```
print(cidades[0][0])
print(cidades[0][1])
print(cidades[1][2])
```

Resultado:

```
São Paulo
Rio de Janeiro
Los Angeles
```

No exemplo a seguir veja como podemos percorrer toda uma matriz, utilizando dois *loops* de repetição, um dentro do outro, com um contador para as linhas e outro para as colunas.

Exemplo:

```
for i in 0..<cidades.count{
    for j in 0..<cidades[i].count{
        print("Cidade: \(cidades[i][j])")
    }
    print("-----")
}
```

Resultado:

```
Cidade: São Paulo
Cidade: Rio de Janeiro
Cidade: Belo Horizonte
Cidade: Espírito Santo
-----
Cidade: Nova Iorque
Cidade: São Francisco
Cidade: Los Angeles
Cidade: Orlando
-----
Cidade: Patópolis
Cidade: Cucamonga
Cidade: Springfield
Cidade: Acapulco
-----
```

```
var cidades:[[String]] = [["São Paulo","Rio de Janeiro","Belo Horizonte","Espírito Santo"],
    ["Nova Iorque","São Francisco","Los Angeles","Orlando"],
    ["Patópolis","Cucamonga","Springfield","Acapulco"]]

print(cidades[0][0])
print(cidades[0][1])
print(cidades[1][2])

print(cidades.count)
print(cidades[0].count)

for i in 0..<cidades.count{
    for j in 0..<cidades[i].count{
        print("Cidade: \(cidades[i][j])")
    }
    print("-----")
}
```

```
Cidade: Nova Iorque
Cidade: São Francisco
Cidade: Los Angeles
Cidade: Orlando
-----
Cidade: Patópolis
Cidade: Cucamonga
Cidade: Springfield
Cidade: Acapulco
-----
```

Chamamos este tipo de procedimento de *nested loop* ou "*loops* aninhados".

Significa que a cada execução do **for**, haverá um outro **for** acontecendo.

Usamos este tipo de procedimento para varrer as **linhas** e **colunas** de uma matriz.

27. Métodos do Array

Conheça algumas das propriedades e métodos que os *arrays* possuem.

a) Atributo *count*

Retorna o número de ítens da lista, ou seja, o **tamanho** do array*.*

Exemplo:

```
var nomes = ["Tito", "Felpudo", "Fofura"];
print(nomes.count)
```

Resultado:

```
3
```

Sendo assim, podemos facilmente acessar sempre o **último item** da lista, utilizando o comando:

Exemplo:

```
var nomes = ["Tito", "Felpudo", "Fofura"];
print(nomes[nomes.count-1])
```

Resultado:

```
Fofura
```

Independentemente do número de itens existentes, se obtivermos o **tamanho da Lista**, podemos subtrair **-1** e saber qual o **Índice do último item da Lista**.

b) Último Item do Array - last

No Swift podemos também utilizar o método **last** para obter o último item do array.

Lembre-se que tratando-se de um *array* vazio (não inicializado), o método **last** vai retornar um objeto nulo. Por isso para utilizar este tipo de método, temos que tratá-lo como uma variável **opcional**, e expor o **conteúdo da variável** através do operador **!**

Exemplo:

```
var nomes:[String] = ["Felpudo", "Fofura", "Lesmo", "Bugado",
"Peluche", "Uruca", "Felpudo Jr."]

print(nomes.last!)
```

Resultado:

```
Felpudo Jr.
```

c) Primeiro Item do Array - first

Podemos acessar também o primeiro item do array, utilizando o método **first**. Isto vai trazer em uma variável **opcional** o valor do primeiro item do *array*

Exemplo:

```
var nomes:[String] = ["Felpudo", "Fofura", "Lesmo", "Bugado",
"Peluche", "Uruca", "Felpudo Jr."]

print(nomes.first!)
```

Resultado:

```
Felpudo
```

Lembre-se que aparentemente esse método faz o mesmo que nomes[0]. Porém lembre-se que ao tentar acessar um índice de um **array vazio** desta maneira, podemos ocasionar um erro. Como o método **first** retorna um objeto **opcional**, fica mais fácil lidar com este tipo de situação.

d) Ordenando os Arrays em Swift

Em Swifto podemos ordenar facilmente as listas com o método **sorted()**. Veja só o exemplo abaixo.

Exemplo:

```
var nomes:[String] = ["Felpudo", "Fofura", "Lesmo", "Bugado",
"Peluche", "Uruca", "Felpudo Jr."]

print(nomes.sorted())
```

Podemos também inverter o array, utilizando o método **reversed().**

Perceba que neste caso, o método reversed precisa depois ser transformado em um objeto do tipo tipo Array.

Para isto basta transformá-lo utilizando o método **Array().**

Exemplo:

```
["Bugado", "Felpudo", "Felpudo Jr.", "Fofura", "Lesmo", "Peluche",
"Uruca"]
```

Exemplo:

```
var numeros = [1,100,30,5,12,15]
print( Array(numeros.sorted().reversed()))
```

Resultado:

```
[100, 30, 15, 12, 5, 1]
```

28. Métodos de Strings

Agora que já conhecemos um pouco sobre arrays e os seus métodos, perceba que os Strings são objetos do tipo lista.

Podemos considerar os Strings como um um **array de caracteres**.

Perceba que também possuem métodos semelhantes aos objetos do tipo array:

a) Método count

Retorna o número de caracteres do **String**, assim como nos outros arrays.

Exemplo:

```
var mensagem = "Oi eu sou o Tito Petri!"
print(mensagem.count)
```

Resultado:

```
23
```

30. Percorrendo uma Lista com for

Agora vamos utilizar um método para percorrer uma lista e acessar cada um dos itens armazenados nesta lista.

Para realizar este procedimento devemos utilizar a **estrutura de repetição for**, que aprendemos anteriormente.

Veja, no exemplo abaixo, como utilizar uma estrutura **for** para percorrer a lista dos argumentos passados pelo console para o array **args**.

Exemplo:

```
var nomes:[String] = ["Felpudo", "Fofura", "Lesmo", "Bugado",
"Peluche", "Uruca", "Felpudo Jr."]

for i in 0..< nomes.count{
    print(nomes[i])
}
```

O **for** irá percorrer a lista do índice (**i**) zero até o último item do array **nomes** imprimindo o valor de cada item no console.

Esta é uma forma bem "artesanal" de fazer um **for** em uma lista.

Aproveite as idéias e conceitos aprendidos aqui, porém saiba que existe no Swift um recurso mais simples para varrer uma lista com um **for**.

Exemplo:

```
for nome in nomes {
    print(nome)
}
```

Podemos percorrer um array com outro tipo de estrutura **for**. Perceba que agora os argumentos da estrutura **for** agora são diferentes.

O **primeiro argumento** é o tipo da variável que vai receber cada item do array (**nome**).

A partir daqui, durante a execução do **for**, cada item será lido e guardado em uma variável chamada **nome,** por exemplo.

O **segundo argumento** depois dos dois pontos (**:**) é o array que você deseja percorrer (**nomes).**

Pare e perceba quantos conceitos envolvidos apenas em uma pequena instrução.

Reflita sobre o tanto de conceitos e idéias envolvidos você precisou entender para interpretar este código.

Muitas destas idéias são regras e práticas universais que você vai usar pra sempre, mesmo se um dia estiver utilizando outra linguagem de programação. Veja que grande habilidade você adquiriu!

31. Projeto #3 - Sorteio de Nomes

Já temos todo o conhecimentos necessário para desenvolver a nossa próxima aplicação.

Um programa para **Sortear Nomes** a partir de uma lista!

Pense e desenvolva um algoritmo que seja capaz de:

1) Obter uma **Lista** de nomes.

2) Sortear um dos nomes da **Lista**.

DESAFIO #8

Desafio você a desenvolver sozinho o Algoritmo do Sorteio de Nomes antes de olhar a minha resolução.

Tentar e errar, esta é a melhor maneira de fixar o conhecimento!

Lesmo

Lembre-se de que existem várias maneiras de desenvolvimento, tente encontrar a sua!

Bugado

DESAFIO #2

Crie um sorteio entre seus amigos no Whatsapp.

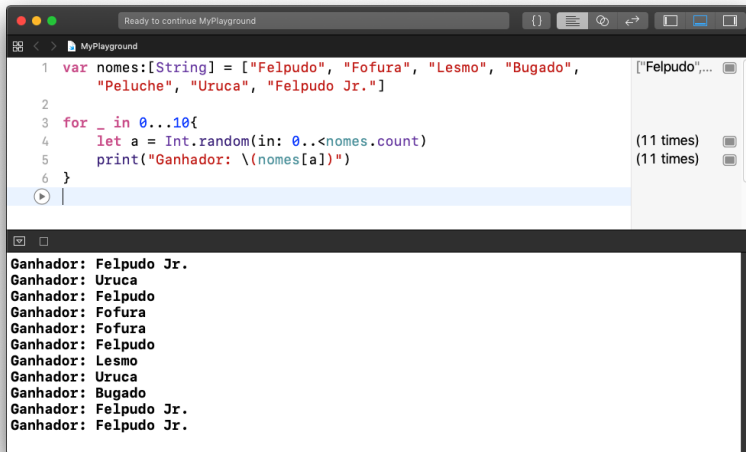
Para executar o sorteio, utilize o algoritmo em Swift que você mesmo criou.

Mostre o seu código para que os participantes vejam que a aplicação é válida.

Veja a seguir a minha resolução para este algoritmo.

Resolução do Projeto #3 - Sorteio de Nomes

```
Ready to continue MyPlayground
MyPlayground

1  var nomes:[String] = ["Felpudo", "Fofura", "Lesmo", "Bugado",
       "Peluche", "Uruca", "Felpudo Jr."]                          ["Felpudo",...
2
3  for _ in 0...10{
4      let a = Int.random(in: 0..<nomes.count)                     (11 times)
5      print("Ganhador: \(nomes[a])")                              (11 times)
6  }
```

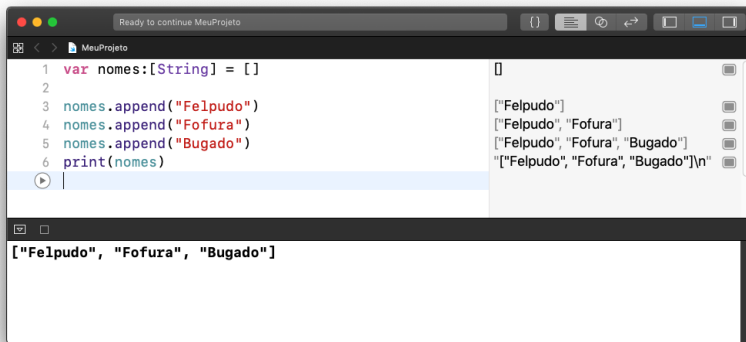
```
Ganhador: Felpudo Jr.
Ganhador: Uruca
Ganhador: Felpudo
Ganhador: Fofura
Ganhador: Fofura
Ganhador: Felpudo
Ganhador: Lesmo
Ganhador: Uruca
Ganhador: Bugado
Ganhador: Felpudo Jr.
Ganhador: Felpudo Jr.
```

Exemplo:

```
var nomes:[String] = ["Felpudo", "Fofura", "Lesmo", "Bugado",
"Peluche", "Uruca", "Felpudo Jr."]

for _ in 0...10 {
    let a = Int.random(in: 0..<nomes.count)
    print("Ganhador: \(nomes[a])")
}
```

Perceba apenas que no algoritmo eu executei o sorteio por 10 vezes.

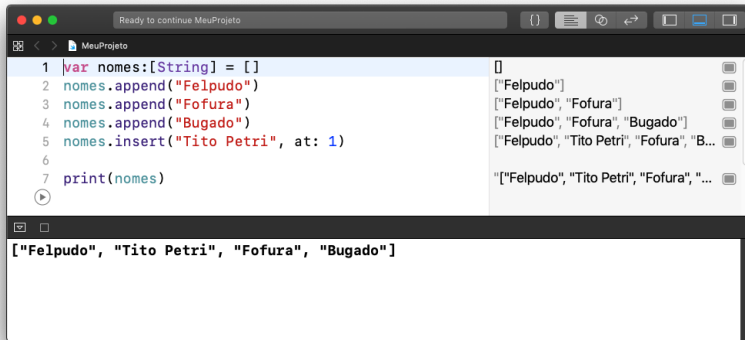
28. Adicionando e Removendo Itens do Array

Vamos agora entender um pouco melhor como manipular os itens dentro do array.

Observe o exemplo:

Exemplo:

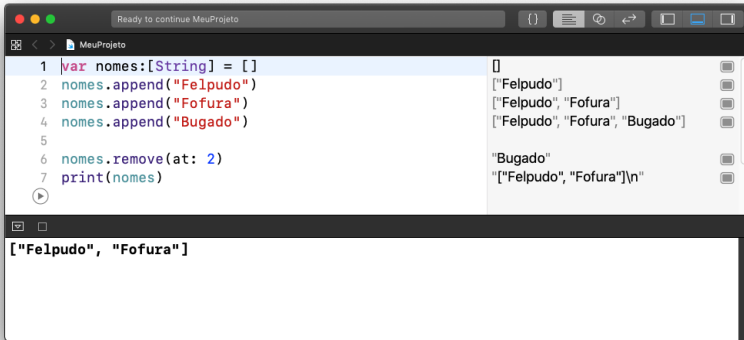
```
var nomes:[String] = []

nomes.append("Felpudo")
nomes.append("Fofura")
nomes.append("Bugado")
print(nomes)
```

Podemos também criar um **array vazio** e ir adicionando os itens ao array progressivamente em nosso código, utilizando o comando **append**.

Podemos também adicionar o item à um determinado índice do array. Por exemplo:

Exemplo:

```
var nomes:[String] = []

nomes.append("Felpudo")
nomes.append("Fofura")
nomes.append("Bugado")

nomes.insert("Tito Petri", at: 1)

print(nomes)
```

O comando **insert**, instrui o compilador a adicionar um determinado objeto a um índice especificado no array. Perceba no exemplo anterior que o String “Tito Petri” foi adicionado logo na segunda casa do array (índice 1).

Da mesma forma, para Removermos um item de uma Lista, é preciso declarar o Comando remove(), indicando o índice (posição) do item que se deseja eliminar através do argumento **at**.

Exemplo:

```
var nomes:[String] = []
nomes.append("Felpudo")
nomes.append("Fofura")
nomes.append("Bugado")

nomes.remove(at: 2)
print(nomes)
```

33. Projeto #4 - Busca por Nome e ID

Neste momento, um belo desafio para você juntar tudo o que vimos até aqui seria montar uma aplicação que busque um nome em uma Lista, associando este nome ao seu respectivo número de registro de matrícula (ID).

Nossa Lista contém os nomes: ["Felpudo", "Fofura", "Lesmo", "Bugado", "Uruca".

Os IDs de cada um são respectivamente: 10, 20, 33, 100, 999.

Utilize ArrayLists para armazenar os Dados e o Comando break para sair da busca, ao encontrar o item.

Para realizar a busca de um nome qualquer, você deverá digitá-lo no campo **command line arguments**, antes de executar a aplicação. Ao final, a aplicação deve dizer se foi ou não encontrado, imprimir o nome pesquisado, e qual o número de identificação (ID) que atribuímos a este nome.

Tente desenvolver esta aplicação e depois confira a minha resolução.

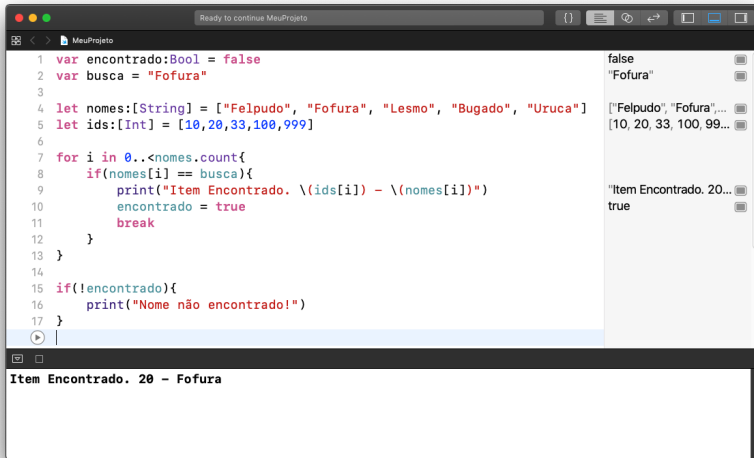
Dicas Importantes:

Utilize **dois Array**, um para armazenar os **nomes** e o outro os **IDs**.

A variável Booleana chamada de **encontrado** servirá para nos indicar que o item foi achado na lista.

Ela permanecerá falsa até que o nome seja encontrado na lista.

Caso nenhum nome seja encontrado, deverá exibir uma **mensagem de aviso**.

Solução para o Projeto #4 - Busca por Nome e ID

```
var encontrado:Bool = false
var busca = "Fofura"

let nomes:[String] = ["Felpudo", "Fofura", "Lesmo", "Bugado",
"Uruca"]
let ids:[Int] = [10,20,33,100,999]

for i in 0..<nomes.count{
    if(nomes[i] == busca){
        print("Item Encontrado. \(ids[i]) - \(nomes[i])")
        encontrado = true
        break
    }
}

if(!encontrado){
    print("Nome não encontrado!")
}
```

34. Métodos e Procedimentos

Métodos são blocos de comandos que usamos para resumir e encapsular uma ação.

Vamos abstrair e imaginar a seguinte situação: Alguém entrando em um carro.

Esta ação envolveria algumas instruções:

- Ir até o carro
- Usar a chave
- Abrir a porta
- Sentar no banco
- Fechar a porta

Podemos resumir todas estas ações chamando apenas um comando: **entrar no carro**. Isto simplificaria muito as coisas no caso da construção de um algoritmo. Cada vez que a **função** ou **método** fosse chamado, todas as instruções relativas à este método seriam executadas.

Veja a seguir o exemplo de como podemos **criar ou registrar uma nova função** em nosso código.

Exemplo:

```
func falar() {
    print("Olá eu sou o Felpudo!")
}
```

Até aqui apenas **criamos (declaramos) o método**. Precisamos agora **executá-lo**. Para isso basta usar o nome do método.

Exemplo:

```
falar()
```

Veja a seguir como ficou a **declaração** e a **utilização** do método que acabamos de criar.

Exemplo:

```
func falar(){
    print("Olá eu sou o Felpudo!")
}

falar()
```

Agora temos o método criado e cada vez que usarmos o comando **falar( )**, o bloco inteiro do método será executado.

Preste atenção em alguns detalhes. Existem dois tipos de métodos. **Com** e **Sem Retorno** de valor.

Podemos também executar métodos passando **valores**, ou **parâmetros** como **argumentos.**

Uma coisa é você instruir o Joãozinho a ir até a padaria. Outra coisa é você instruí-lo a ir na padaria e comprar **5 pães** e **1 leite.**

Podemos entender que os **pães** e **leite** são os argumentos do método, os parâmetros que foram dados para que o método fosse executado.

Veja a seguir como seria a utilização de um método com **passagem de argumentos**.

Exemplo:

```
func correr(distancia:Float){
    print("Correu \(distancia) metros.")
}
```

Agora, declaramos qual será o argumento necessário para executar o método (**Float distância**).

Portanto, para executar o método Falar() precisaremos sempre passar um **argumento** ou **parâmetro** do tipo String entre os parênteses.

Observe um segundo exemplo, agora iremos passar **dois argumentos** para o método ser executado.

Exemplo:

```
func correr(distancia:Float){
    print("Correu \(distancia) metros.")
}

correr(distancia: 50.7)
```

Perceba que na hora da declaração (criação) do método, definimos o **tipo** da **variável** e o **nome**, separando cada **argumento** por vírgula.

35. Função ou Método com Retorno

Até então, criamos métodos que apenas são executados e acabam ali.

Existem também o que chamamos de **função**, é basicamente o mesmo que um método, porém a **função** nos devolve algum valor como resultado final da sua execução.

Até então, sempre declaramos nossos métodos usando a palavra **void**, isto significa que o **retorno** da função é **vazio**, ou seja, ela **não tem retorno.**

Observe agora como é uma função, ou método com retorno de valor.

Exemplo:

```
func multiplica (x:Float, y:Float) -> Float{
    return x*y
}

var resultado = multiplica(x: 3, y: 10)
print(resultado)
```

Perceba que depois da passagem dos argumentos, utilizamos o operador **->** seguido do **tipo** de valor que esta função deve retornar através do comando **return**.

Se você executar este código, verá que nada vai acontecer, a função **Multiplicar()** agora apenas executa a multiplicação, e retorna o resultado, que vai ser guardado na variável **resultado**.

Exemplo:

```
func multiplica (x:Float, y:Float) -> Float{
    return x*y
}

var resultado = multiplica(x: 3, y: 10)
print(resultado)
```

Lembre-se então:

a) Método
É o encapsulamento de uma sequência de ações. Funções e Procedimentos são considerados Métodos!

b) Função
É um método que necessariamente **retorna um valor**. Precisa declarar o tipo que será retornado pela função (como em nosso exemplo: **-> Float**) e deve haver o comando **return** dentro do método.

c) Procedimento
É o método que **não tem retorno**, no caso do Swift, que não utiliza o operador **->**.

É comum os desenvolvedores não darem atenção a estes detalhes, e saírem chamando tudo de **função**. Ponto para você que agora você já sabe exatamente qual é a diferença!

36. Projeto #5 - Validação do CPF

Existe uma norma para se definir a geração de números do CPF. É um **Algoritmo** já bem conhecido entre os programadores e estudantes de Ciência da Computação.

Trata-se de ler os 9 primeiros dígitos e calcular os dois últimos dígitos da sequência total de 11 números. Vamos entender como é feito este cálculo:

Vamos utilizar o CPF 123.456.789. Os dígitos finais para que este CPF seja válido devem ser **0** e **9**.

Vamos entender os passos para este algoritmo funcionar:

1) Primeiro, precisamos multiplicar os valores dos 9 digitos pelos números de 10 a 2 como na grade abaixo:

CPF	1	2	3	4	5	6	7	8	9
x	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*
=	**10**	**18**	**24**	**28**	**30**	**30**	**28**	**24**	**18**

2) Agora some todos os resultados e obtenha o **Resto da Divisão** por **11**.

Se o número for **menor que 2**, o **Primeiro Dígito** é considerado o número **Zero**!

Se o número **for 2 ou mais**, subtraia o resultado de **11**.

Exemplo:

Soma Total	Divisão	Resto da Divisão	Primeiro Dígito
210	11	1	0

Agora repita o processo de soma total, **inserindo o primeiro dígito** encontrado ao final da sequência, e fazendo a **multiplicação de 11 até 2** desta vez.

Exemplo:

CPF	1	2	3	4	5	6	7	8	9	0
x	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*
=	**11**	**20**	**27**	**32**	**35**	**36**	**35**	**32**	**27**	**0**

Veja como fica o resultado final da validação deste CPF.

Exemplo:

Soma Total	Divisão	Resto da Divisão	Segundo Dígito
255	11	9	9

Agora, o que sabemos é que baseado nos nove primeiros dígitos do CPF (Ex.: 123456789) podemos calcular qual terá que ser os últimos dois dígitos para que este CPF seja válido. No caso, descobrimos que são o **0** e o **9**.

É muito importante que você entenda o funcionamento do algoritmo muito bem antes de começar a sua programação.

Aconselho que você faça antes os cálculos em um **pedaço de papel**.

Escreva os valores no formato de uma tabela como aqui no Livro e faça os cálculos para chegar no resultado.

Estes cálculos manuais chamamos de **Teste de Mesa**. Eles são muito importantes para nos assegurarmos do funcionamento do algoritmo antes de partirmos para os códigos e a programação.

Use o seu CPF ou de alguns amigos e faça alguns testes.

Depois, confira a seguir a minha resolução para este algoritmo.

Resolução para o Projeto #5 - Validação do CPF

```
var numeros:[Int] = [1,2,3,4,5,6,7,8,9,0,0]
var multiplica = 10;
var soma = 0;
var digito = 0;

for i in 0..<9{
    soma = soma + (numeros[i] * multiplica)
    multiplica -= 1
}

// Calculando o Primeiro Dígito
digito = soma%11

if(digito<2){
    numeros[9] = 0
}else{
    numeros[9] = 11 - digito
}

multiplica = 11
soma = 0

```

```
123.456.789-09
```

```

// Calculando o Segundo Dígito
for i in 0..<10{
    soma = soma + (numeros[i]*multiplica)
    multiplica-=1
}

digito = soma%11
if(digito<2){
    numeros[10] = 0
}else{
    numeros[10] = 11 - digito
}

// Exibindo o CPF Completo e Formatado
var cpfFinal = ""
for i in 0..<numeros.count {
    if((i>0)&&(i%3==0)&&(i<9)){cpfFinal+="."}
    if(i==9){cpfFinal+="-"}
    cpfFinal += String(numeros[i])
}
print(cpfFinal)
```

```
123.456.789-09
```

```
var numeros:[Int] = [1,2,3,4,5,6,7,8,9,0,0]
var multiplica = 10;
var soma = 0;
var digito = 0;

for i in 0..<9{
    soma = soma + (numeros[i] * multiplica)
    multiplica -= 1
}

// Calculando o Primeiro Dígito
digito = soma%11

if(digito<2){
    numeros[9] = 0
}else{
    numeros[9] = 11 - digito
}

multiplica = 11
soma = 0

// Calculando o Segundo Dígito
for i in 0..<10{
    soma = soma + (numeros[i]*multiplica)
    multiplica-=1
}

digito = soma%11
if(digito<2){
    numeros[10] = 0
}else{
    numeros[10] = 11 - digito
}

// Exibindo o CPF Completo e Formatado
```

```
var cpfFinal = ""
for i in 0..<numeros.count {
    if((i>0)&&(i%3==0)&&(i<9)){cpfFinal+="."}
    if(i==9){cpfFinal+="-"}
    cpfFinal += String(numeros[i])
}
print(cpfFinal)
```

37. Projeto #6 - Números Primos

Os **Números Primos** são números inteiros **divisíveis** apenas **por 1** e **por eles mesmos!**

Os números 2, 3, 5, 7, 11, 13 são exemplos de **números primos** pois são divisíveis apenas por **eles mesmos** ou **por 1**.

DESAFIO #1

Eu te desafio a utilizar tudo o que você aprendeu até aqui e Desenvolver um algoritmo para calcular todos os números primos de zero até um determinado número inserido na linha de comando do compilador.

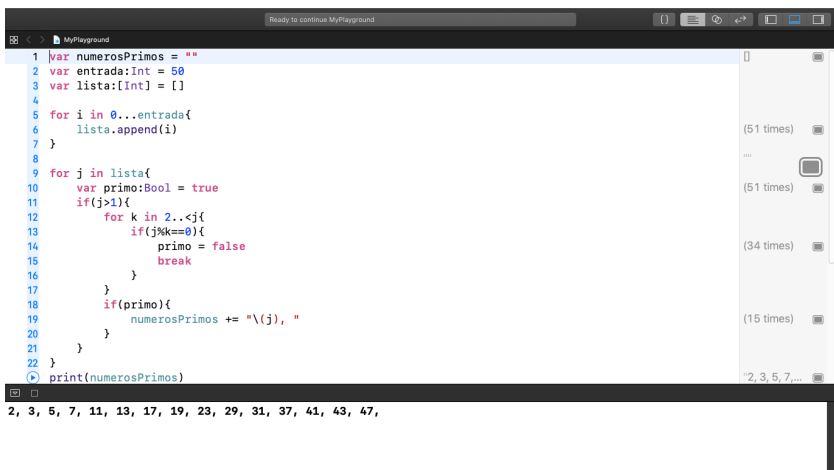
Lesmo

Para resolver este desafio, você vai ter que varrer uma lista do número zero até o número inserido pelo usuário e dividi-lo por todos os números menores que ele.

Para fazer esta verificação do meu exemplo eu resolvi utilizando dois *loops* de repetição, um dentro do outro, assim como fizemos no capítulo *"2. Array multidimensional ou Matriz"* para varrer uma **tabela ou matriz de dados.**

Aproveite e observe a foto e a piada a seguir.

```
1  for i in (1...10).reversed(){
2      for _ in 1...i{
3          print("*", terminator: "")
4      }
5      print("", terminator: "\n")
6  }
```

Caso você não entenda a *gag* ou *meme* ou piada à seguir, sugiro que estude mais os laços de repetição, ou então reformule seu senso de humor.

Observação: o argumento (terminator:""**) serve apenas para continuar imprimindo na mesma linha do console. Não expliquei isso anteriormente para não confundir o leitor. Porém esta é uma boa hora para você entender mais este comando da linguagem Swift =).*

Já vimos anteriormente como utilizar loops aninhados para percorrer matrizes. O conceito aqui é parecido.

Agora que você já entendeu bem sobre o **aninhamento de *loops*** ou ***nested loops*** (conceito de um *loop* de repetição **for** dentro do outro).

Observe a seguir a minha resolução para o problema dos números primos.

Resolução para o Projeto #6 - Números Primos

Ready to continue MyPlayground

MyPlayground

```
1  var numerosPrimos = ""
2  var entrada:Int = 50
3  var lista:[Int] = []
4
5  for i in 0...entrada{
6      lista.append(i)
7  }
8
9  for j in lista{
10     var primo:Bool = true
11     if(j>1){
12         for k in 2..<j{
13             if(j%k==0){
14                 primo = false
15                 break
16             }
17         }
18         if(primo){
19             numerosPrimos += "\(j), "
20         }
21     }
22 }
   print(numerosPrimos)
```

(51 times)
(51 times)
(34 times)
(15 times)
"2, 3, 5, 7,...

```
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
```

```
var numerosPrimos = ""
var entrada:Int = 50
var lista:[Int] = []

for i in 0...entrada{
```

```
    lista.append(i)
}

for j in lista{
    var primo:Bool = true
    if(j>1){
        for k in 2..<j{
            if(j%k==0){
                primo = false
                break
            }
        }
        if(primo){
            numerosPrimos += "\(j), "
        }
    }
}
```

Para executar o código acima **insira um número na linha de argumento** e compile a aplicação.

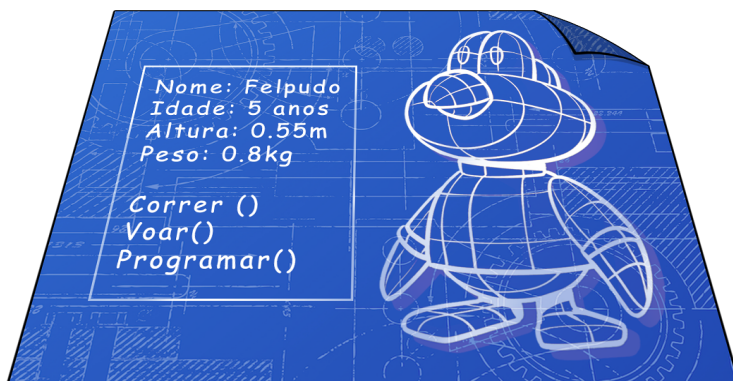
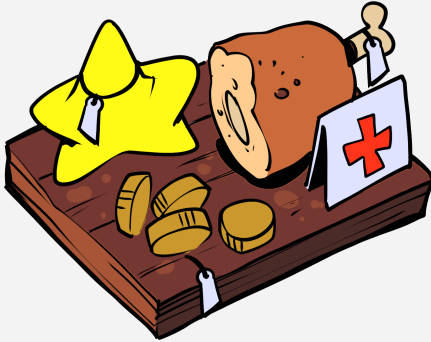
O resultado obtido no console será todos os números primos de 0 até o seu número digitado.

Resultado para Primos de 0 a 100:

```
0, 1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47
```

38. Classes no Swift

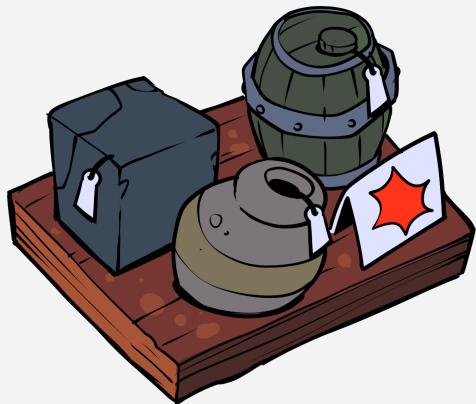
As **classes** são estruturas que servem de modelos para criarmos objetos.

Imagine uma receita, um molde, que reúne todas as características e métodos à respeito da formação de um determinado **objeto.**

A **classe** é praticamente o modelo que reúne os **métodos** e **atributos** refrentes à um determinado **objeto.**

Cada objeto tem suas próprias características (atributos, variáveis) e métodos de funcionamento.

Ao criar ou registrar a classe, você define todos estes métodos e atributos uma única vez.

Depois, você pode reutilizar estes métodos e atributos novamente. A cada objeto criado na sua programação, ele já virá com todos estas propriedades.

Para "fabricar" um objeto dentro da nossa programação precisamos primeiro especificar sua **classe**.

Com base nesta classe podemos criar **novos objetos**.

A seguir, vamos criar e utilizar uma nova classe no Swift.

Exemplo:

```
class Animal{
    var nome:String?
    var idade:Int?
    var peso:Float?
    var cor:String?
}
var animal:Animal = Animal()

animal.nome = “Felpudo”
animal.idade = 12
```

A partir do comando **class** definimos uma nova classe e criamos os parâmetros e atributos dentro das chaves **{...}**.

Exemplo:

```
class Animal{
    var nome:String?
    var idade:Int?
    var peso:Float?
    var cor:String?

    func correr(){
        print("Correu.")
    }

    func falar(mensagem:String){
        print("Ola eu sou o: \(mensagem)")
    }
}

var animal:Animal = Animal()
```

```
animal.correr()
animal.falar(mensagem: "Oi eu sou o Felpudo!")
```

E para instanciar (ou criar) um objeto na sua programação utilize o método construtor, e declare o objeto seguindo as regras que utilizaria para criar uma variável qualquer.

Primeiro o **tipo**, depois o nome da variável, e depois sua inicialização (onde teremos que utilizar o construtor **new).**

Veja, a seguir o exemplo completo, como criar um novo objeto, e manipular seus métodos e atributos.

Exemplo:

```
class Animal{

    var nome:String?
    var idade:Int?
    var peso:Float?
    var cor:String?

    func correr(){
        print("Correu.")
    }

    func falar(mensagem:String){
        print("Ola eu sou o: \(mensagem)")
    }
}

var animal:Animal = Animal()
```

```
animal.nome = “Felpudo”
animal.idade = 12

animal.correr()
animal.falar(mensagem: "Oi eu sou o Felpudo!")
```

Perceba que o objeto foi criado e posteriormente foi inserido os valores nos seus atributos.

Também podemos utilizar o **Método Construtor** para já criar e parametrizar o objeto, logo no comando da criação.

Para isto, na sua classe você deve utilizar o método construtor da classe.

Exemplo:

```
 init(nome:String, idade:Int, peso:Float, cor:String) {
     self.nome = nome
     self.idade = idade
     self.peso = peso
     self.cor = cor
  }
```

Observe que através deste método, que deve ser um método público e ter o mesmo nome da classe, estamos passando quatro argumentos como parâmetros. Estes argumentos serão utilizados para inicializar os atributos da classe, que são referenciados por **.self**.

Agora, na hora de criar o objeto, utilizamos o seguinte método.

Exemplo:

```
var animal:Animal = Animal(nome: "Felpudo", idade: 12, peso: 50,
cor: "verde")
```

Assim, seu objeto já está criado e inicializado.

Confira o código inteiro da implementação da classe já com o seu **Método Construtor**.

Exemplo:

```
class Animal{

    var nome:String?
    var idade:Int?
    var peso:Float?
    var cor:String?

    init(nome:String, idade:Int, peso:Float, cor:String) {
        self.nome = nome
        self.idade = idade
        self.peso = peso
        self.cor = cor
    }

    func correr(){
        print("Correu.")
    }

    func falar(mensagem:String){
        print("Ola eu sou o: \(mensagem)")
```

```
    }
}

var animal:Animal = Animal(nome: "Felpudo", idade: 12, peso: 50,
cor: "verde")

print(animal.nome!)
print(animal.idade!)
print(animal.peso!)
print(animal.cor!)

animal.correr()
animal.falar(mensagem: "Oi eu sou o Felpudo!")
```

39. Hierarquia de Classes

Hierarquia é o conceito de parentesco (pai e filho) que criamos entre as classes.

Uma classe pode herdar todos os parâmetros e atributos de uma classe superior.

Vamos usar de exemplo um ser humano.

No caso, um objeto do tipo Humano poderia compartilhar de alguns atributos comuns a outros animais, porém ele pode também ter suas próprias atribuições e funções específicas.

Observe como podemos estender uma classe como no caso anterior.

Exemplo:

```
class Humano : Animal {

    var profissao:String?
    var carro:String?

    init(nome:String, idade:Int, peso:Float, cor:String,
profissao:String, carro:String) {

        self.profissao = profissao
        self.carro = carro
        super.init(nome: nome, idade: idade, peso: peso, cor: cor)
    }

    override func correr() {
        print("Correu 1.000 Metros")
    }

}


var humano:Humano = Humano(nome: "Tito", idade: 12, peso: 50,
cor: "azul", profissao: "Programador", carro: "Fusca")

print(humano.nome)
humano.correr()
```

Criamos uma nova classe, e usamos o comando **extend** para estender todos os parâmetros e métodos da classe referida.

No método construtor podemos utilizar o comando **super.init** para inicializar os parâmetros da classe superior (no caso a classe Animal).

Agora, a **Classe Animal** tem seus determinados atributos e métodos, e a **Classe Humano** tem todos os atributos e métodos da classe animal, e ainda seus próprios atributos e métodos.

Entender este conceito de hierarquias facilita muito a estruturação da programação. E vai tornar nossa programação muito mais simples, organizada e fácil de ser reutilizada.

Perceba que em um game, ou aplicativo, cada janela, componente, objeto, e personagem pode ser organizado como uma classe para a facilitação do entendimento do código e o reaproveitamento das programações.

40. Modificadores de Acesso

São formas de **ocultar** ou **expor** os **Atributos** de **Métodos** das classes.

a) Public

A variável pode ser vista e modificada de todas as outras classes que podem acessar o objeto.

No **modificador público** todos têm acesso.

b) Private

A variável só poderá ser vista e modificada de dentro da **classe** em que foi criada.

A única **Classe** que tem acesso ao **Atributo** é a própria classe que o define.

Ou seja, se uma classe "Pessoa" declara um **Atributo Privado** chamado "nome", somente a classe "Pessoa" terá acesso a ele.

c) Default

Um atributo **default** (identificado pela **ausência de modificadores**) pode ser acessado por todas as classe que estiverem no mesmo pacote em que a classe que possui o atributo se encontra.

d) Protected

Esse **Modificador** é o que mais causa confusão, ele é praticamente igual ao **default**, com a diferença de que se uma classe (mesmo que esteja fora do pacote) estende alguma classe com o modificador **protected**, esta classe poderá ter acesso a este atributo.

O acesso de um atributo protegido é dado por **pacote** e por **herança**.

42. Sobrescrita e Sobrecarga de Método

Aqui entram dois conceitos bem importantes na hora de criar os Métodos de uma hierarquia de classes:

a) Sobrescrita de Método - @Override

O Operador **@Override** é uma forma de garantir que você está sobrescrevendo um método, e não criando um novo.

Perceba que, no exemplo abaixo, a subclasse (filho) substituiu o método da superclasse (pai).

Exemplo:

```
public class ClassePai {
    public void imprime() {
        print("Mensagem Pai!")
    }
}

public class ClasseFilho extends ClassePai {
    @Override
    public void imprime() {
        print("Mensagem Filho!")
    }
}
```

O que causa um pouco de confusão é que este operador é opcional. Se você removê-lo perceberá que não vai mudar nada!

Este operador existe para reforçar o entendimento da estrutura dos métodos nas classes.

Em alguns compiladores, se você usar um operador **@Override** em um método que não existe na classe superior, ele pode dar erro!

b) Sobrecarga de Método - Overload

Quando dizemos que um Método é Sobrecarregado (*Overloaded Method*) significa que este método é utilizado para diferentes funções, de acordo com os parâmetros que receber.

Veja que, na classe do exemplo abaixo, definimos três métodos com o mesmo nome. Porém, cada um deles recebe argumentos de diferentes tipos.

De acordo com o argumento passado, o compilador já entende qual é o método relacionado.

43. Modelo de Dados

A partir de agora, vamos entender um pouco mais sobre a **arquitetura de dados** de um programa.

Vou utilizar como exemplo o cadastro dos dados de uma pessoa. Podemos reunir todos estes dados em um objeto único, estruturando seus atributos no formato de um **modelo de dados**.

Para isso vamos definir uma classe, que vou chamar de **Contato**.

Cada objeto desta classe vai guardar todos os atributos ou **dados de uma pessoa**, que eu preciso armazenar. Basicamente, será uma classe com alguns atributos.

A única novidade é que agora vamos usar os ArrayLists para armazenar uma lista de **objetos** criados a partir da classe **Contato.**

Exemplo:

```
import Swift.util.List;
import Swift.util.ArrayList;

class Contato{

String nome;
  String telefone;
  String email;
  String empresa;

  public Contato(String nome, String telefone, String email, String
empresa){
    this.nome = nome;
    this.telefone = telefone;
    this.email = email;
    this.empresa = empresa;
  }
}

public class OlaMundo
{
  public static ArrayList<Contato> minhaLista = new
ArrayList<Contato>()
  public static void main(String[] args)
  {
    Contato meuContato = new Contato("Tito
Petri","555-1234","tito.petri@gmail.com","Tito's Coorp.")

    print(meuContato.nome)
    print(meuContato.telefone)
    print(meuContato.email)
    print(meuContato.empresa)

    minhaLista.add(meuContato)

    print(meuContato)
```

```
    print(minhaLista.get(0))
  }
}
```

Resultado:

```
Tito Petri
555-1234
tito.petri@gmail.com
Tito's Coorp.
Contato@6d06d69c
Contato@6d06d69c
```

Perceba que qualquer objeto do mundo real possui uma série de características atribuídas a ele (nome, cor, tamanho, peso, idade entre outros).

Podemos analisar estas características e modelar o nosso banco de dados a partir delas.

44. Projeto #7 - Cadastro e Busca

Neste próximo algoritmo, nossas programações já começam a ficar maiores e o nível de abstração aumenta ainda mais.

Serei breve na explicação, para que o próprio leitor pare, concentre-se e tente entender por si códigos que já foram explicados isoladamente, ao longo de todo este material.

No programa a seguir, basicamente vamos criar um ArrayList de objetos da classe **Contato** e inicializá-lo com alguns contatos.

Observe que também existe a implementação de alguns métodos importantes para trabalhar no Banco de Dados:

- Adicionar Contato
- Remover Contato por Nome
- Buscar Contato
- Imprimir Lista Completa

Exemplo:

```
// Declaração da Classe Contato
class Contato{
    var nome:String?
    var sobrenome:String?
    var email:String?
    var telefone:String?

    init(_ nome:String,_ sobrenome:String,_ email:String,_
telefone:String){
        self.nome = nome
        self.sobrenome = sobrenome
        self.email = email
        self.telefone = telefone
    }
}

// Criando Objetos da Classe Contato
var contatoA:Contato =
Contato("Tito","Petri","tito.petri@gmail.com","555-1234")
var contatoB:Contato =
Contato("Felpudo","Júnior","felpudo@mail.com","555-0000")
var contatoC:Contato = Contato("Fofura","da
Silva","fofura@mail.com","555-0000")

// Declaração do Método para Exibir um Contato
func imprimeContato (_ contato:Contato){
    print("Nome: \(contato.nome!) \(contato.sobrenome!)\ne-Mail:
\(contato.email!)\nTelefone: \(contato.telefone!)\n")
}

// Utilizando o Método para Exibir um Contato
imprimeContato(contatoA)
imprimeContato(contatoB)
imprimeContato(contatoC)

// Criando uma Lista de Objetos da Classe Contato
```

```
var listaContatos:[Contato] = []

// Adicionando Objetos Contato a Lista
listaContatos.append(contatoA)
listaContatos.append(contatoB)
listaContatos.append(contatoC)
listaContatos.append(Contato("Uruca", "McMoney",
"uruca@mail.com", "555-1313"))

// Percorrendo a Lista e Exibindo os Objetos da Classe Contato
for item in listaContatos{
    imprimeContato(item)
}

func imprimeNomes(contatos:[Contato]){
    for item in contatos {
        print(item.nome!)
    }
}

func imprimeNomeComEmail(contatos:[Contato]){
    for item in contatos {
        let meuContato:Contato = item as Contato
        print("Nome: \(meuContato.nome!) - email:
\(meuContato.email!)")
    }
}

imprimeNomes(contatos:listaContatos)

// Método para Buscar um Contato na Lista

func buscaContato(contatos:[Contato], nomeBuscado:String){
    var encontrado:Bool = false
    for item in contatos {
        let meuContato:Contato = item as Contato
```

```
        if meuContato.nome! == nomeBuscado {
            print("Nome Encontrado!!!\n...")
            print(meuContato.nome!)
            print(meuContato.email!)
            print(meuContato.telefone!)
            encontrado = true
        }
    }

    if !encontrado {
        print("Nome Não Encontrado!")
    }
}

buscaContato(contatos:listaContatos, nomeBuscado:"Tito")

// Método para Apagar um Contato da Lista

func apagarContato(contatos:[Contato], nomeBuscado:String){
    var encontrado:Bool = false

    for i in 0...contatos.count-1 {

        if contatos[i].nome! == nomeBuscado {
            print("Contato Apagado!!!\n...")
            print(contatos[i].nome!)
            listaContatos.remove(at: i)
            encontrado = true
        }
    }

    if !encontrado {
        print("Nome Não Apagado!\n...")
    }
}

apagarContato(contatos:listaContatos, nomeBuscado:"Tito")
```

```
// Exibindo toda a Lista

for item in listaContatos{
    imprimeContato(item)
}
```

Uruca

DESAFIO FINAL

Agora todo o conhecimento que obtivemos durante este Livro será aplicado na construção deste programa.

Escreva e Compile linha por linha, método por método, até entender todos os conceitos e idéias por trás da construção de toda esta aplicação de cadastro de contatos.

42. Banco de Dados

O que criamos anteriormente é uma tabela relacional de valores. Pense que toda a estrutura de dados de um programa ou *site* pode estar contida em uma tabela desta.

Além de gravar nomes e dados de usuários, podemos salvar as preferências do usuário durante a navegação em um *site* ou aplicativo.

Ordenar dados da partida de um jogo, atributos da construção de objetos visuais ou gráficos.

Os Bancos de Dados mais modernos são ferramentas que, além de adicionar e remover itens em tabelas relacionais, oferecem-nos métodos mais sofisticados para organizar e ler estas tabelas.

Exemplo:

NOME	TELEFONE	EMAIL	EMPRESA
Bugado	555-6666	bugado@mail.com	ACME Inc.
Felpudo	1234	contato@mail.com	Escola do VideoGame
Tito Petri	555-1234	tito.petri@gmail.com	Tito Coorp.
Fofura	1234	contato@mail.com	Escola do VideoGame
Lesmo	999-8888	lesmo@mail.com	ACME Inc.
Uruca	555-6666	uruca@mail.com	ACME Inc.

A tabela anterior está totalmente desordenada. Imagine se você pudesse ordená-la por cada um dos atributos, veja quanta coisa interessante poderia descobrir.

Abaixo podemos ordenar todos os itens do Banco, de acordo com a ordem alfabética do campo NOME por exemplo.

Exemplo:

NOME	TELEFONE	EMAIL	EMPRESA
Bugado	555-6666	bugado@mail.com	ACME Inc.
Felpudo	1234	contato@mail.com	Escola do VideoGame
Fofura	1234	contato@mail.com	Escola do VideoGame
Lesmo	999-8888	lesmo@mail.com	ACME Inc.
Tito Petri	555-1234	tito.petri@gmail.com	Tito Coorp.
Uruca	555-6666	uruca@mail.com	ACME Inc.

Ou agora, ordenar pelo número de telefone. Só de visualizar a tabela conseguimos entender quais são os usuários que possuem o mesmo telefone, por exemplo.

Exemplo:

NOME	TELEFONE	EMAIL	EMPRESA
Felpudo	1234	contato@mail.com	Escola do VideoGame
Fofura	1234	contato@mail.com	Escola do VideoGame
Tito Petri	555-1234	tito.petri@gmail.com	Tito Coorp.
Bugado	555-6666	bugado@mail.com	ACME Inc.
Uruca	555-6666	uruca@mail.com	ACME Inc.
Lesmo	999-8888	lesmo@mail.com	ACME Inc.

E assim, podemos enxergar nossa lista por várias ordenações diferentes, como o email.

Exemplo:

NOME	TELEFONE	EMAIL	EMPRESA
Bugado	555-6666	bugado@mail.com	ACME Inc.
Felpudo	1234	contato@mail.com	Escola do VideoGame
Fofura	1234	contato@mail.com	Escola do VideoGame
Lesmo	999-8888	lesmo@mail.com	ACME Inc.
Tito Petri	555-1234	tito.petri@gmail.com	Tito Coorp.
Uruca	555-6666	uruca@mail.com	ACME Inc.

E a empresa, por exemplo.

Exemplo:

NOME	TELEFONE	EMAIL	EMPRESA
Bugado	555-6666	bugado@mail.com	ACME Inc.
Lesmo	999-8888	lesmo@mail.com	ACME Inc.
Uruca	555-6666	uruca@mail.com	ACME Inc.
Felpudo	1234	contato@mail.com	Escola do VideoGame
Fofura	1234	contato@mail.com	Escola do VideoGame
Tito Petri	555-1234	tito.petri@gmail.com	Tito Coorp.

Se você fosse programar a listagem de itens de um *site* ou aplicativo, perceba agora como poderia ser útil este tipo de recurso.

Hoje em dia existem muitas ferramentas de **Banco de Dados**. Entre elas, posso sugerir duas muito úteis e famosas.

SQL é uma linguagem de programação específica para **Bancos de Dados**. Muitas linguagens de programação conversam com **SQL**, e poderiam se integrar facilmente a um **Banco de Dados** em **SQL,** e utilizá-lo em *sites* e aplicativos *iOS* ou *iOS*, por exemplo.

Firebase é o que chamamos de **BackEnd de Dados**. O **Firebase** é um *site*/serviço onde você pode criar e consultar **Bancos de Dados** com muita facilidade.

Como ainda estamos dando nossos primeiros passos em **Swift** e utilizando um compilador um pouco limitado em recursos, não existe uma integração simples com o **SQL**.

Porém, quando você der os próximos passos e começar a programar seus aplicativos para o sistema **iOS** por exemplo (**utilizando o IDE Xcode**), você já terá entendido todos os conceitos de base necessários para se trabalhar com o **Swift**, e as ideias principais sobre **Bancos de Dados**.

43. Armazenamento de Dados

Quando criamos uma variável, ela fica armazenada temporariamente na memória do computador. A partir do momento em que você fechar o *browser* ou a janela do compilador, esse dado se apaga.

Porém, existem maneiras de armazenar os dados permanentemente na memória. É o que chamamos de **Dados Persistentes**. Perceba que quando seu computador ou celular é desligado e ligado novamente, todos os dados (fotos, vídeos, arquivos) ainda permanecem lá.

Isto é porque eles foram armazenados permanentemente em uma memória sólida ou rígida - *Hard Disk*. Um tipo de memória em que o dado fica gravado para sempre.

Diferente da memória RAM, onde todos os dados são zerados quando a máquina desliga.

Podemos armazenar os dados permanentemente de algumas maneiras.

Gravando na memória sólida (*HD* ou disco rígido). Todo equipamento possui uma memória pois é lá onde está instalado o sistema operacional, ou o sistema principal da máquina (Windows, Mac OS, iOS ou iOS).

Gravando em algum cartão de armazenamento, como o do seu celular, um *pendrive* ou um HD externo ao seu computador.

Gravando o dado na nuvem. Como hoje a conexão com a *internet* já é quase que constante, muitas aplicações gravam o dado direto em algum computador/servidor que esteja conectado através da *internet*.

Ou seja, o seu aplicativo se conecta ao servidor e manda o dado para ser armazenado em algum computador específico (servidor).

Aqui, no compilador *online* de **Swift,** também seria uma tarefa bem trabalhosa armazenar o **Dado** permanentemente na aplicação. Porém, quando for trabalhar com os **IDEs** como o Xcode, você perceberá que esta tarefa é feita com muita facilidade.

44. Introdução ao Xcode

Até aqui nós já utilizamos o Xcode, porém apenas para testar os códigos no Playground, utilizando a Linguagem Swift.

Perceba que com o que conhecemos, ainda não é possível se criar uma aplicação para os Sistemas da Apple (MacOS, iPhones, iPads, tvOS e Apple Watch).

Para chegar neste nível, ainda teremos que conhecer um monte de Ferramentas, Recursos e Bibliotecas do Xcode IDE.

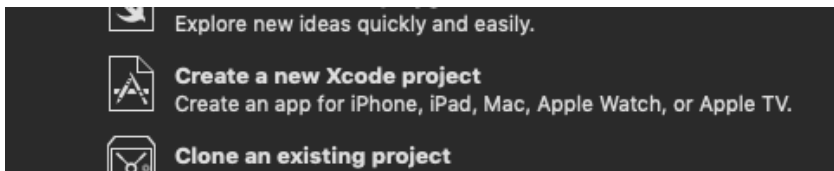
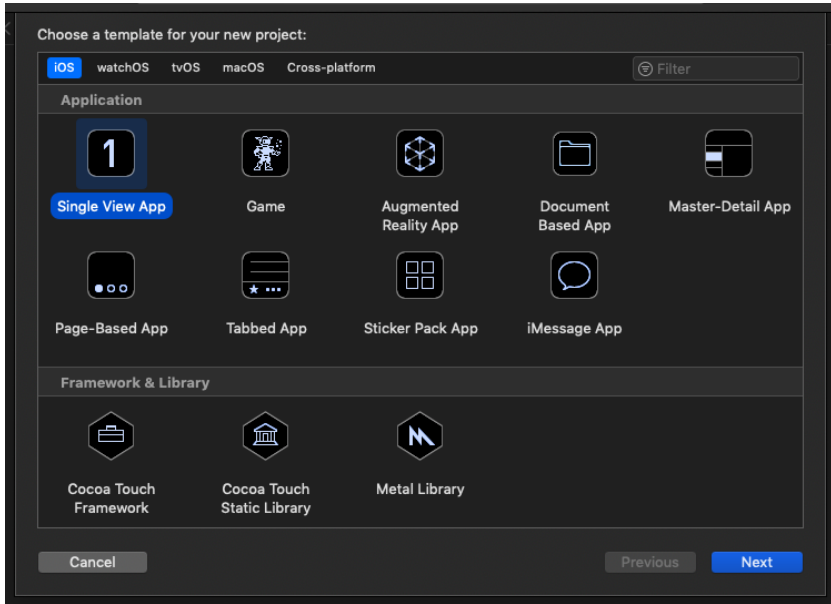
O Xcode é o programa que vai te permitir reunir tudo o que você já conhece sobre a Linguagem Swift, aos recursos dos aparelhos iOS (como câmera, acelerômetro, internet, redes sociais e tudo o que é possível de se realizar com um dispositivo iOS por exemplo).

A partir desta seção, vou te ensinar os primeiros passos no Xcode para você aprender o básico e depois assistir às mais de 500 vídeo-aulas que eu tenho publicadas sobre o assunto.

46. Meu Primeiro Aplicativo iOS

Então vamos lá, fecha o seu Playground e reinicie o Xcode.

Na tela inicial do programa,

escolha agora a opção **Create a New Xcode Project**

Perceba que esta é a opção pra criar projetos de aplicações para iPhone, Mac, AppleWatch e Apple TV.

Deixe selecionadas as opções **iOS** e **Single View App**. e Clique em **Next**.

Agora dê um nome para o seu projeto. Chamarei o meu de **Meu Primeiro App**.

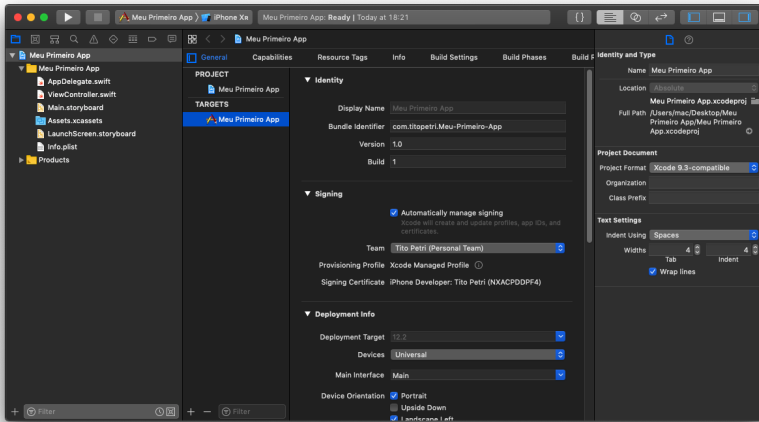
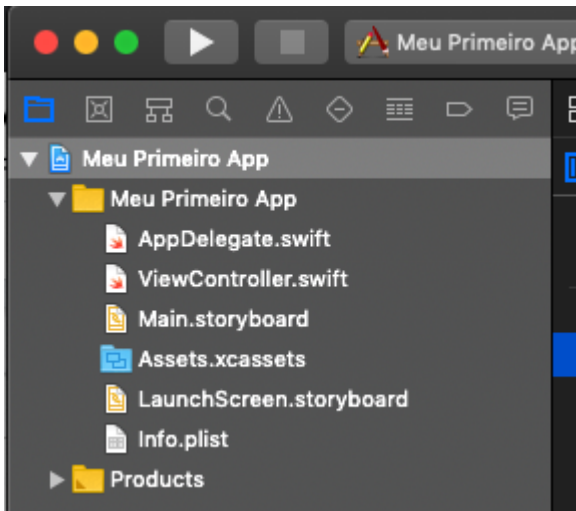
No campo **Organization Identifier:** coloque o endereço reverso do seu website. No meu caso que é www.titopetri.com.br, vou utilizar apenas com.titopetri.

Isto é apenas uma maneira de garantir que cada aplicativo publicado na Apple Store futuramente, tenha uma identificação única.

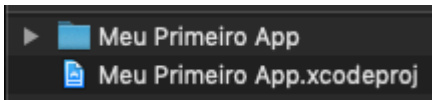
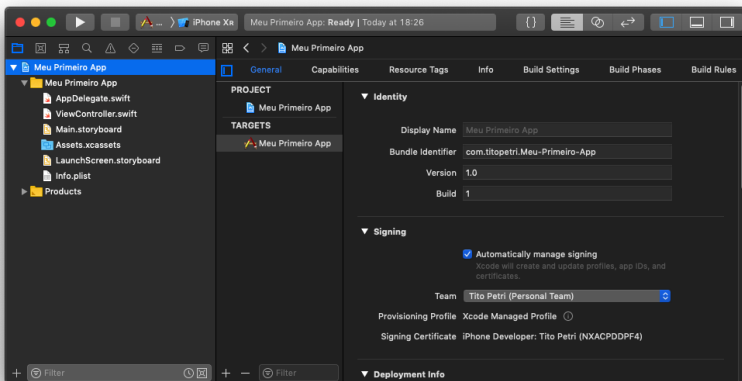
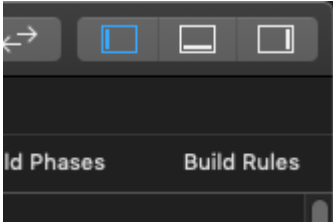
Ao clicar novamente em **Next**, selecione uma pasta para criar e guardar o seu projeto.

Perceba que agora o Xcode vai criar nesta pasta, uma série de sub-pastas e arquivos.

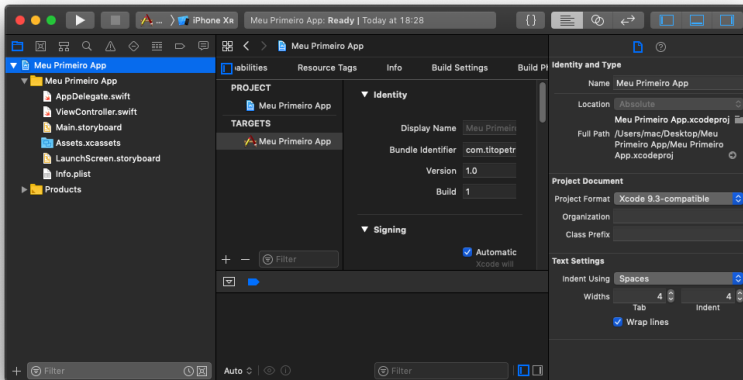
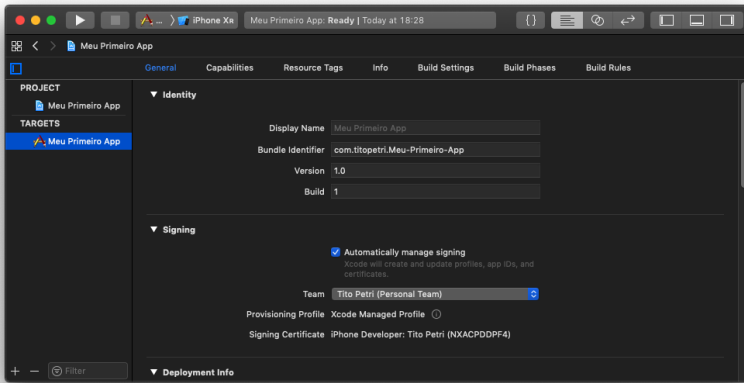
Para programar nosso aplicativo iOS, teremos que conhecer vários destes arquivos e o que cada um deles configura ou executa no app.

Esta é a tela inicial do Projeto do Xcode

Perceba que no menu lateral esquerdo, conseguimos navegar entre os mesmos arquivos encontrados no diretório do projeto.

O Xcode é uma espécie de explorador/browser que te permite navegar e visualizar os arquivos e códigos do projeto.

Vamos agora conhecer os principais arquivos do nosso projeto do Xcode.

O arquivo **.xcodeproj** é a raíz do nosso projeto, clicando duas vezes nele, voce abre o projeto no Xcode.

Selecionando este arquivo dentro do Xcode, você será levado às configurações básicas do projeto.

Repare também no canto superior direito

Estes três botões mostram e escondem os três menus laterais do Xcode: **Navegador**, **Console de Debug** e **Inspetor**

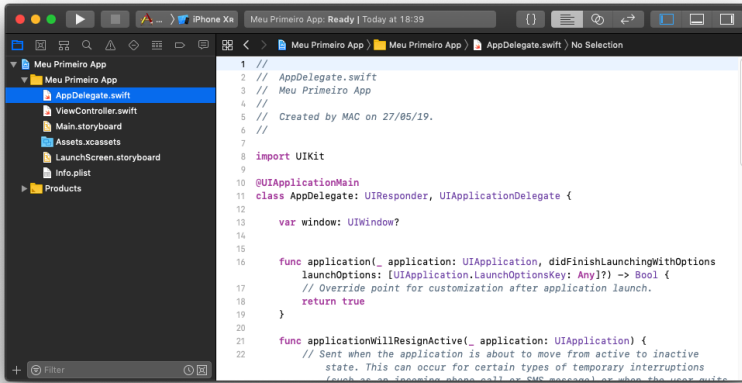
Você pode ligar ou desligar esses botões de forma a esconder ou mostrar os menus.

Exibindo os menus:

Escondendo os Menus

Existem 3 atalhos bem úteis para este comando:

CMD+0 - Navegador
CMD+ALT+0 - Inspector
CMD+SHIFT+Y - Console de Debug

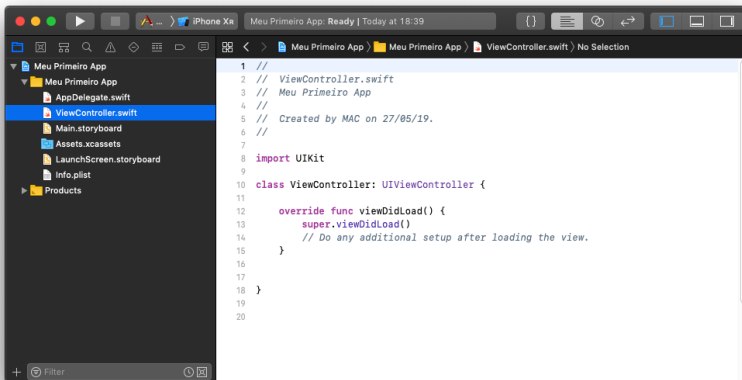
Agora pelo Navegador, vamos navegar através dos princiapis arquivos de um Projeto do Xcode:

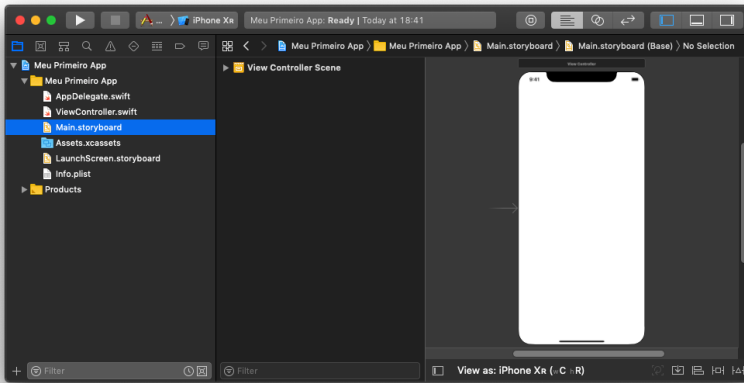
AppDelegate.swift

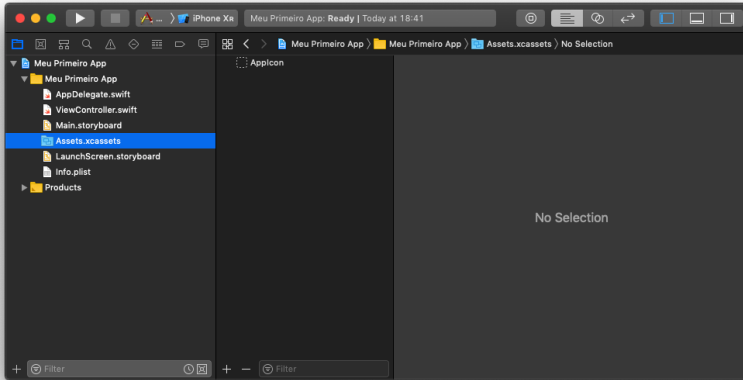
Este arquivo é a raiz de todos os códigos do projeto. Aqui podemos encontrar os eventos que são disparados ao se abrir, fechar ou suspender a aplicação iOS.

ViewController.swift

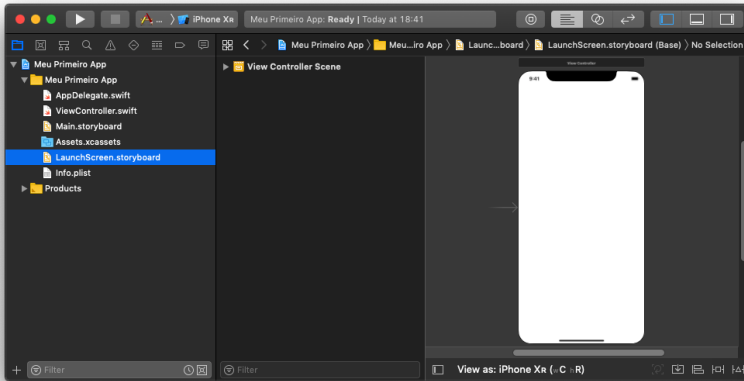
A ViewController é o código associado à primeira tela da nossa aplicação que em nosso caso ela ainda está em branco.

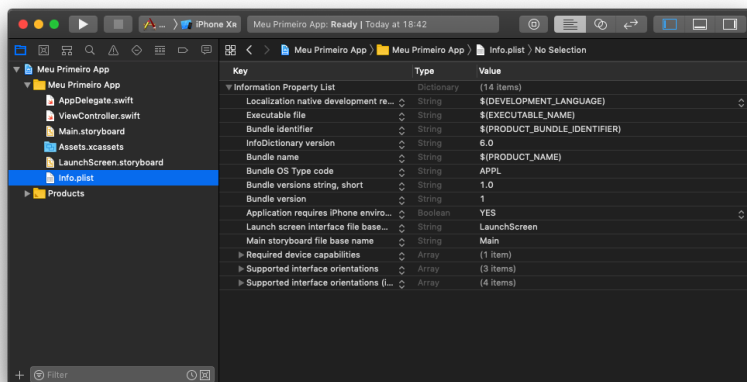
É aqui que vamos escrever nosso programa em Swift. Veremos adiante.

```
//
//  ViewController.swift
//  Meu Primeiro App
//
//  Created by MAC on 27/05/19.
//

import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.
    }


}
```

Main.storyboard

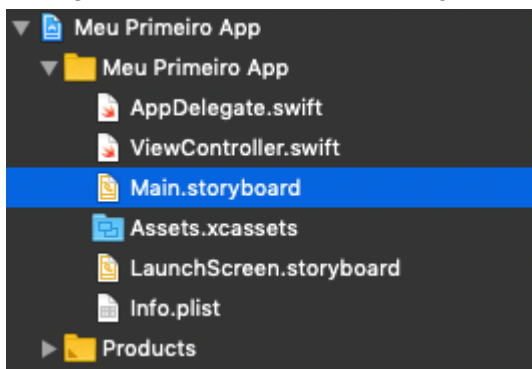
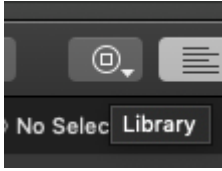
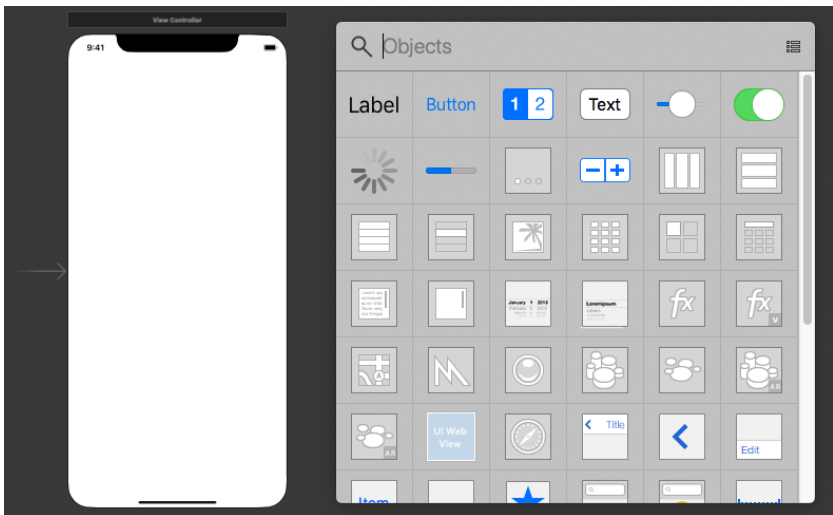
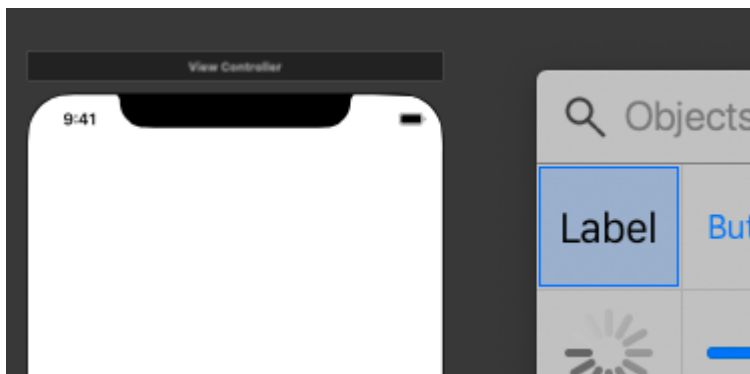
O Storyboard é o fluxograma de ligação e navegação entre as telas da aplicação. Por aqui podemos criar novas telas e utilizar os componentes nativos para o iOS como Botões, Textos, Imagens, Listas (TableViews), Sliders e vários outros.

Assets.xcassets

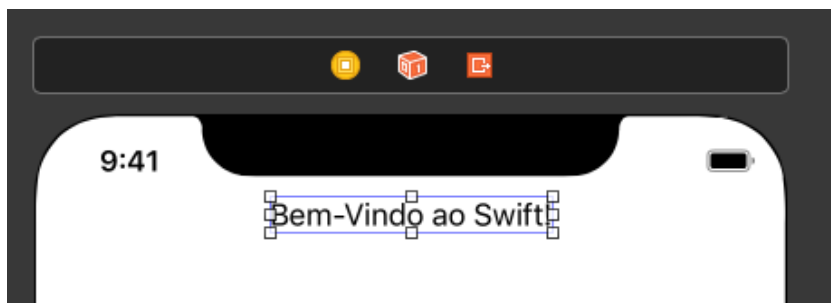
Onde guardaremos os recursos gráficos (como ícones e arquivos de imagens) da nossa aplicação.

LaunchScreen.storyboard

Esta tela é exibida instantaneamenta ao abrir a aplicação. É a tela de início ou carregamento do app.

Info.plist

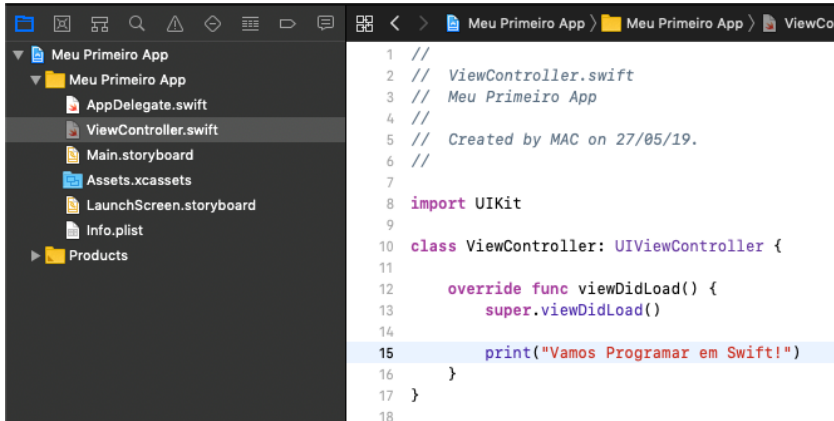
Aqui configuramos recursos (como a utilização de outras línguas ou fontes de texto) e permissões para utilizar a internet, ou a câmera do aparelho por exemplo.

Por enquanto, vamos apenas executar nosso primeiro aplicativo iOS no Simulador.

Antes vamos apenas fazer alguma pequena modificação em alguns arquivos que já conhecemos.

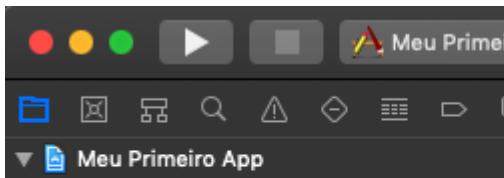
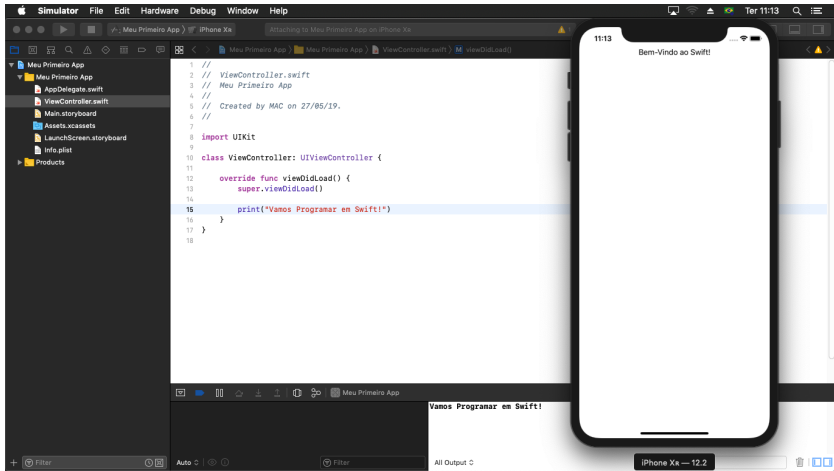
Navegue até o arquivo **Main.Storyboard**

No canto superior direito do Xcode, você vai encontrar o botão **Library**.

Por ela você terá acesso a uma paleta de **Componentes Nativos do iOS**. São componentes visuais como botões, caixas de texto e outros objetos que utilizaremos para montar o layout do nosso aplicativo.

Vamos começar pelo mais simples, o **Label**, a caixa de texto.

Selecione e arraste um **Label** da paleta **Library** para cima da tela em branco da nossa **ViewController**.

Dando dois cliques sobre o texto, você consegue editá-lo.

Agora navegue até o arquivo **ViewController.swift**

```
Meu Primeiro App
  Meu Primeiro App
    AppDelegate.swift
    ViewController.swift
    Main.storyboard
    Assets.xcassets
    LaunchScreen.storyboard
    Info.plist
  Products
```

```swift
//
//  ViewController.swift
//  Meu Primeiro App
//
//  Created by MAC on 27/05/19.
//

import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        print("Vamos Programar em Swift!")
    }
}
```

Insira um comando **print** dentro do método **viewDidLoad()**.

Este método será executado assim que a **ViewController** for carregada.

Vamos agora **compilar** e **executar** a aplicação no Simulador iOS. Basta clicar no botão **Run / Play** no canto superior direito.

Agora aguarde até o Simulador iOS abrir e executar sua aplicação. Isto pode demorar até alguns minutos.

Pronto, já temos nosso primeiro Aplicativo iOS funcionando.

Agora você já tem todo o embasamento sobre a Linguagem Swift e o Xcode IDE para iniciar sua jornada no desenvolvimento de aplicações iOS.

À seguir, vou te direcionar a mais de 400 video aulas de Swift e Xcode para voce dominar processo desde o zero até a criação e publicação de Aplicativos, Games e Realidade Aumentada na AppStore.

45. Linguagens de Programação

Existem muitas ferramentas e linguagens de programação diferentes.

Frequentemente, os alunos me perguntam qual é a melhor ferramenta, ou qual linguagem aprender?

Sugiro que esta pergunta seja sempre trocada para:

"***Que tipo de aplicação você gostaria de desenvolver?"***

Baseado nesta pergunta, escolha uma ferramenta ou linguagem que vai te dar os recursos para desenvolver este seu produto.

Veja as principais ferramentas e linguagens utilizadas para diferentes tipos de desenvolvimento.

Exemplo:

Páginas, Web Sites e Sistemas para Internet:
HTML
CSS
Php
SwiftScript

** Lembrando que o HTML e CSS teoricamente não são linguagens de programação, e sim de marcação, pois servem apenas para montar e configurar as páginas, e não para montar algoritmos. Porém essas linguagens e conhecimentos devem estar presentes no dia-a-dia do Desenvolvedor Web.*

Aplicativos *Mobile* e *Desktop*, recomendo as seguintes linguagens e IDEs.

Exemplo:

Desktop & Mobile Apps	IDE	Plataforma
Swift	Xcode	iOS
Swift	Xcode	Mac OS e iOS
C-Sharp (C#)	Visual Studio	Windows

E, no caso do desenvolvimento de *Games*, Realidade Virtual e Realidade Aumentada, recomendo as seguintes ferramentas.

Exemplo:

Games e VR	IDE	Plataforma
Swift	Xcode	Mac OS e iOS
C-Sharp (C#)	Unity	Multiplataformas

Não existe uma linguagem ou ferramenta melhor ou pior, o principal é entender as diferenças e conhecer qual delas vai te ajudar a chegar no produto que você deseja criar.

Lembre-se que o primeiro passo sempre será *"Entender qual é o produto que você deseja saber criar!"*...

Parabéns querido aluno por chegar até aqui e ter adquirido mais este valioso conhecimento!

Se quiser aprender sempre mais sobre criação de Jogos e Aplicativos, não deixe de conhecer o Aprenda Programar, meu portal de cursos online onde você pode se especializar em:

- Algoritmos e Lógica de Programação
- Modelagem e Animação 3D
- Criação de Personagens para Jogos e Filmes
- Programação de Aplicativos Nativos para iOS e Android
- Criação de Games 2D, 3D e Realidade Virtual
- Realidade Aumentada e Visão Computacional
- Metodologia STEAM
- Robótica e Impressão 3D

Para virar aluno do Aprenda Programar você deve se inscrever pela plataforma Hotmart, no link abaixo.

Adquira seu acesso para sempre ao Aprenda Programar:
https://hotmart.com/product/en/aprenda-programar-com-tito-petri